Newcastle University

School of Electrical and Electronic Engineering

Design of Asynchronous Microprocessor for Power Proportionality

by Maxim Rykunov

PhD Thesis

December 2013

Contents

List of Figures

List of Tables

Abstract

Microprocessors continue to get exponentially cheaper for end users following Moore's law, while the costs involved in their design keep growing, also at an exponential rate. The reason is the ever increasing complexity of processors, which modern EDA tools struggle to keep up with. This makes further scaling for performance subject to a high risk in the reliability of the system. To keep this risk low, yet improve the performance, CPU designers try to optimise various parts of the processor. Instruction Set Architecture (ISA) is a significant part of the whole processor design flow, whose optimal design for a particular combination of available hardware resources and software requirements is crucial for building processors with high performance and efficient energy utilisation. This is a challenging task involving a lot of heuristics and high-level design decisions. Another issue impacting CPU reliability is continuous scaling for power consumption. For the last decades CPU designers have been mainly focused on improving performance, but "keeping energy and power consumption in mind". The consequence of this was a development of energy-efficient systems, where energy was considered as a resource whose consumption should be optimised. As CMOS technology was progressing, with feature size decreasing and power delivered to circuit components becoming less stable, the energy resource turned from an optimisation criterion into a constraint, sometimes a critical one. At this point power proportionality becomes one of the most important aspects in system design. Developing methods and techniques which will address the problem of designing a power-proportional microprocessor, capable to adapt to varying operating conditions (such as low or even unstable voltage levels) and application requirements in the runtime, is one of today's grand challenges. In this thesis this challenge is addressed by proposing a new design flow for the development of an ISA for microprocessors, which can be altered to suit a particular hardware platform or a specific operating mode. This flow uses an expressive and powerful formalism for the specification of processor instruction sets called the Conditional Partial Order Graph (CPOG). The CPOG model captures large sets of behavioural scenarios for a microarchitectural level in a computationally efficient form amenable to formal transformations for synthesis, verification and automated derivation of asynchronous hardware for the CPU microcontrol. The feasibility of the methodology, novel design flow and a number of optimisation techniques was proven in a full size asynchronous Intel 8051 microprocessor and its demonstrator silicon. The chip showed the ability to work in a wide range of operating voltage and environmental conditions. Depending on application requirements and power budget our ASIC supports several operating modes: one optimised for energy consumption and the other one for performance. This was achieved by extending a traditional datapath structure with an auxiliary control layer for adaptable and fault tolerant operation. These and other optimisations resulted in a reconfigurable and adaptable implementation, which was proven by measurements, analysis and evaluation of the chip.

Acknowledgements

Thought only my name appears on the cover of this thesis, a great number of people have contributed to its success.

First and foremost, I would like to express my sincere gratitude to my supervisor, Prof. Alex Yakovlev. I have been amazingly fortunate to have such a patient, enthusiastic and trustworthy advisor with immense knowledge, which he is available to share on a 24/7 basis. Alex was the one introduced me to world of asynchronous systems to me and guided me throughout my PhD research. I am also grateful to Prof. Vladimir Davydov, who was my undergraduate supervisor in Saint-Petersburg State Polytechnical University in Russia.

My sincere thanks also goes to my co-advisor, Dr. Albert Koelmans, who has been always there to give a good piece of advice. I am also grateful to him for consistent notation in my writings and for carefully reading and commenting on countless revisions of this and other publications throughout the research.

I am thankful to another my good friend and a person whose PhD research motivated me to develop novel approaches in microprocessor design – Andrey Mokhov. His encouragement, practical advice and insightful discussions about the research helped me to overcome many difficult situations and finish this dissertation.

My special thanks goes to Danil Sokolov, Reza Ramezani, Arseniy Alekseyev, Fei Xia, Delong Shang, Alex Bystrov, Raa'ed Aldujaily, and other colleagues in the Microelectronics System Design Group for insightful comments, constructive criticisms and help.

I am also indebted to all the laboratory technicians in our school for their practical advice and many insightful discussions and suggestions, in particularity to Darren Mackie, who helped us a lot at the stage of the developing of the demonstrator PCB.

Lastly but most importantly, none of this would have been possible without the love, support and patience of my family. My family and my fiancé, Alena, to whom this dissertation is dedicated to, has been always there for me and believed in me even when I myself could not. Many thanks to all my friends for their support, motivation and inspiration.

Finally, I appreciate the financial support by EPSRC grant EP/I038357/1 (eFuturesXD, project PowerProp) that funded research on power-proportionality and EPSRC Impact Acceleration Account project "Dataflow Computation a la Carte" EP/K503885/1 that supported prototyping and commercialisation activities.

Chapter 1

Introduction

Since 2007 our society has used more energy for browsing the Internet than for air travel [79]. It is also predicted that the energy (and environmental) footprint of computation and data traffic will steadily increase in the near future: data centres will grow and so will the network infrastructure, together with the number of terminal nodes of the global information network such as computers, mobile phones, gadgets and other connected cyber-physical devices (the so called Internet of Things). Energy-efficiency of components at all levels of the computation hierarchy is thus becoming a major concern for the microelectronics industry. A serious factor impeding progress in addressing this concern is a wide gap between the ways in which energy efficiency is approached by hardware and software engineers, and this gap is matched by a lack of mutual understanding between the two communities.

To address this issue we discuss an approach to bridge this gap by developing a shared design criterion, called *power-proportionality*, on the basis of which both electronics and programming solutions can be judged. A computing system, for it to be considered power-proportional, has to keep power consumption and computation load proportional to each other [163]. That is, an idle system would ideally consume no power, whereas, given a small energy budget, the system would respond by reducing its computation flow and reduce the delivered Quality-of-Service (QoS), and still remain functional. The state-of-the-art systems have a generally poor power-proportionality; for example, the

servers used in data centres typically consume 50-60% of peak power under 10% of peak load [98]. Figure. 1.1 depicts the idea of energy-proportionality and its relation to the delivered QoS. The left plot represents the notion of energy-proportional computing [18], where in a real design there is a particular level of minimum energy per operation, which does not decrease no matter how low the activity of the design goes. However, energy-proportionality is a property in which even at small amounts of energy level some useful activity can still be generated (the optimal design). One can look at this chart from a different angle (see the right hand-side plot), where the activity level axis is shown as the delivered QoS and the energy per action represented as power level. From this point of view, QoS in a real design decreases much quicker than its power level, and therefore the design effort is to increase the delivered QoS in low power supply conditions.

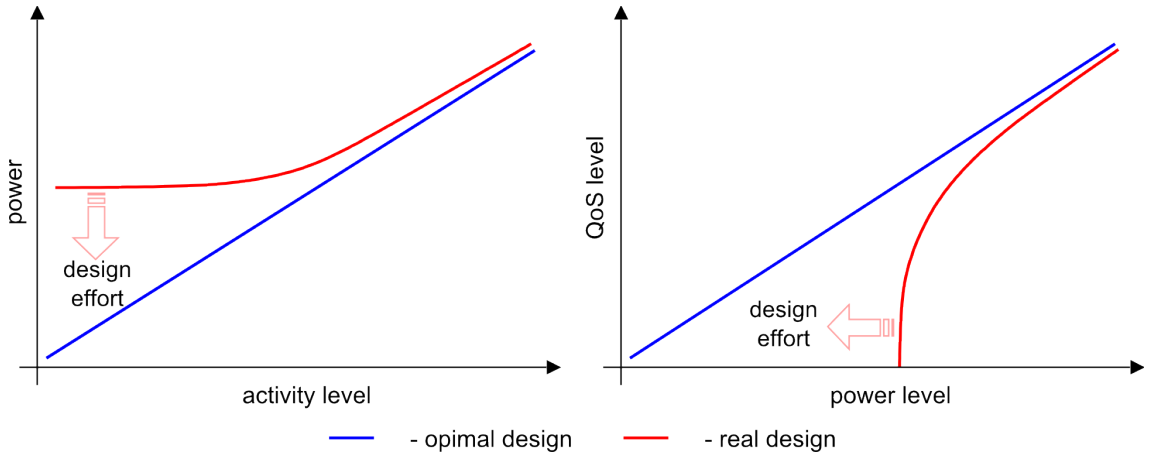


Figure 1.1: Power-proportionality versus power-efficiency

Often it is very difficult to design both power-proportional and energy-efficient systems at the same time. Figure 1.2 shows two power-proportional designs: one (Design 1) is more energy-efficient at low power levels, while the other is more efficient at high power levels. If one wants to build a system that is both power-proportional and power-efficient in a wide range of supply voltages, the best way is to build a hybrid solution, which combines the strengths of both designs. Hence Design 1 could be used in idle mode and Design 2 in full power mode.

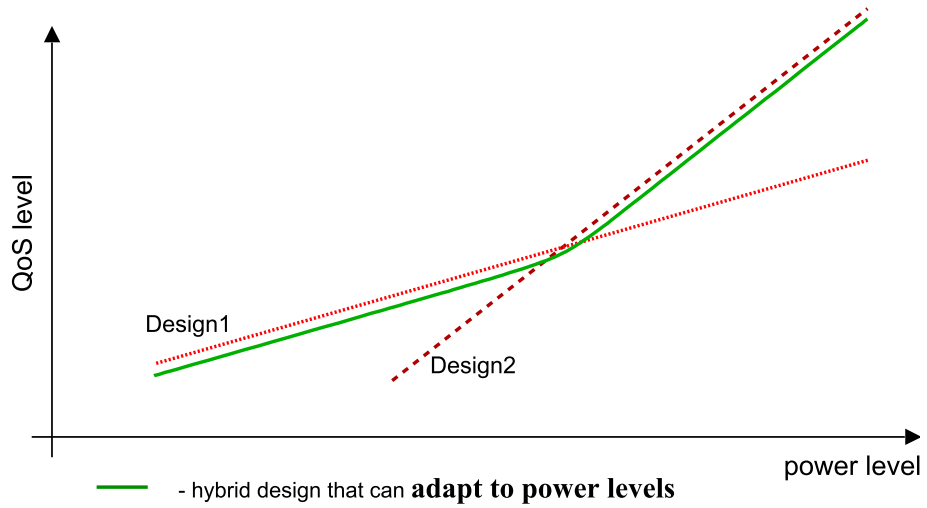


Figure 1.2: Two different power-proportional design

So why are modern systems not power-proportional? We hypothesise that this is because they are designed to operate in a narrow range of conditions and are typically optimised for either high performance or low power consumption. This approach is inherently flawed because all the design effort is focused on one particular operation mode. Hence Integrated Circuits (ICs) developers diversify from general-purpose processing with a single CPU into the realm of System on Chips (SoCs) with multiple specialised cores combined in a single design to efficiently serve a range of predefined applications. Such functional diversification is also motivated by production costs, design reuse, productivity issues and time-to-market constraints. This trend has given rise to the development of application-specific processing cores [91]. At design time, application-specific cores are just instantiated with little or no effort on their joint optimisation, e.g., for better utilisation of hardware components or for improved reliability of the whole system. All functional adaptation to the needs of a particular application is made at run time, by activating one (or a few) of the specialised cores. An example of such a heterogeneous architecture is Cell microprocessor developed by the Sony-Toshiba-IBM alliance [64]. It combines a general-purpose core of modest performance with eight high-performance coprocessing elements. The general-purpose core runs an OS and schedules tasks onto the specialised cores to accelerate multimedia applications.

In addition to functional diversification there is a significant demand for systems

that operate in a wider spectrum of operating conditions (in terms of performance, energy consumption, reliability, etc.). This non-functional diversification is apparent in a recently announced ARM's big.LITTLE architecture [14]. It couples a low-power core with a high-performance core in order to dynamically adjust the computation resource and power consumption of the system. Nvidia extends this approach to multi-core processors by introducing a low-power "companion" core to its quad-core Tegra series of SoCs [119]. The companion core is manufactured using a low-power silicon process and operates at a low clock rate, while the four main cores are performance-oriented. In both ARM and Nvidia architectures, the running state can be quickly transferred between the cores, thus efficiently switching between the low-power and high-performance modes.

The existing techniques split the functionality over several processor cores. That is mostly driven by pressures of productivity, backward compatibility issues and design reuse requirements. Intertwining the functional and non-functional diversities in a heterogeneous system results in a huge design exploration space for all combinations of operating modes and system functionalities. It is very hard to meet the time-to-market demands by considering the cores of such a system individually.

In light of the above, CPU engineers currently focus on the design and implementation of power-proportional microprocessors, capable of adapting to varying operating conditions (such as low and/or unstable power voltage level) and application requirements (mode of software execution) in runtime.

1.1 Power proportionality and reconfigurability

How can one build a reconfigurable and adaptable processor? The first thought could be to implement it in *Field-Programmable Gate Array* (FPGA) [58] technology, allowing static and, sometimes, dynamic reconfiguration. Such a processor would be capable of adjusting its internal structure or behaviour by rewiring the interconnections between its components or even by changing its functionality at the level of individual components. This technology can provide fine-grained control over a system at runtime. However the associated overheads are extremely high. In particular, in terms of energy consump-

tion, FPGAs are typically more expensive than *Application-Specific Integrated Circuits* (ASICs).

Since the fine-grain reconfigurability offered by FPGAs is overly costly, one must consider the coarse-grain reconfigurable architectures in the ASIC realm. They significantly lower the overheads by dropping reconfigurability in datapath components and utilising custom designed versions instead. The control logic and interconnect fabric, however, retain the capability to reconfigure. The key design and implementation challenge is to formally describe and synthesise a controller whose task is to coordinate *hard system resources* (the datapath components and interconnects) according to the runtime information on the availability of *soft system resources* (energy, time); the latter can also include information on hardware faults in a system, thereby allowing the controller to bypass faulty components whenever possible.

A conventional approach to the specification and synthesis of control logic is to employ *Finite State Machines* (FSMs) [118] or interpreted *Petri Nets* (PNs) [50] as an underlying modelling formalism. Within this approach the designer explicitly describes the controller's behaviour for each combination of available resources and operating conditions. The number of such combinations and corresponding behaviours grows exponentially with the size and degree of adaptability of the system. This leads not only to the *state space explosion problem*, but also to the explosion of the specification size [107], thus slowing down the synthesis tools, reducing productivity, and increasing the overall cost of ASIC development.

Our approach is based on the crucial observation that *the controller's behaviours are strongly related to each other* in different operating conditions. Indeed, when a system configuration is changed incrementally, e.g., a datapath component goes offline and another is used in its place, the overall behaviour of the controller is affected in the same incremental manner, hence it is inefficient to separate these two similar behaviours in different specifications, and one would want to have a joint configuration.

It has been demonstrated that the FSM and PN formalisms are not well-suited to describing families of many related behaviours [107] and the design methodologies based

on them have poor scalability in the context of reconfigurable systems. As an alternative, the *Conditional Partial Order Graph* (see Section 2.2) model was introduced in [107]. This model enables us to specify and synthesise the whole processor as a homogeneous system, but still retain the ability to change its behaviour in order to meet the application's functional requirements and adjust to the selected operating mode.

Power-proportionality can also be tackled from a system-level point of view. In the next section we address the main digital-logic design philosophies in the aspect of power-modulated computing.

1.2 Asynchronous approach in power-proportional design

At the system-level, there are three general digital-logic design approaches: the (clocked) synchronous [84], the (clock-free) asynchronous [111, 139] and the hybrid globally asynchronous locally synchronous (GALS) [37].

In synchronous systems, operations between circuits are synchronised to a global timing reference (a global clock). The clock frequency in these systems is set in such a way that the operational time of all its components should be within the period of the clock, hence the highest clock frequency of the whole system is defined by the critical path of its slowest block. In order to improve performance and energy consumption[1], the clock frequency of the system needs to be adjusted (e.g. Multi-level Voltage Scaling (MVS), Dynamic Voltage and Frequency Scaling (DVFS) [129], etc.) to match the critical path of is components, as the environment conditions vary along with specific operating modes. Figure 1.3 (The figure was taken from analysis of gate delay timing variability on a various voltage supply for a 90nm process [33]) shows that synchronous designs operate only on a narrow voltage range and gate timing variability, hence they are not providing robust computations in variable operating conditions.

[1] With a current multi-billion transistor design distribution of a global clock in the entire system could be costly in terms of area and power (up to 40% of the total chip power consumed by the clock distribution network [54]).

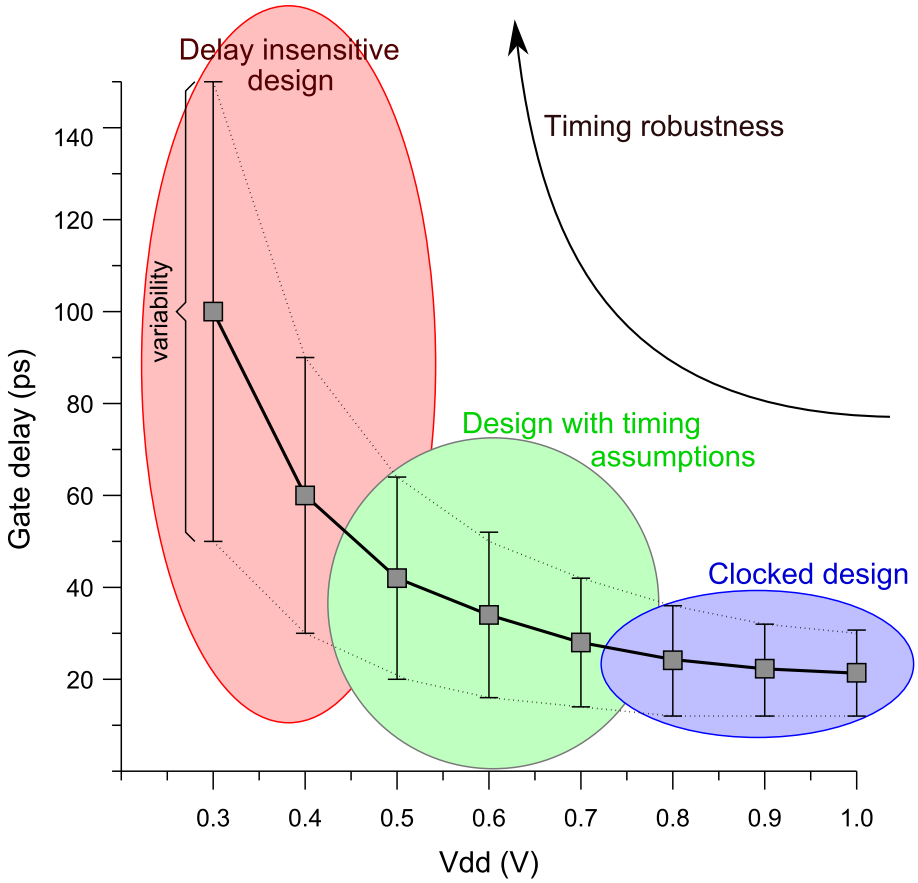


Figure 1.3: Gate delay variability versus voltage supply.

Asynchronous (or self-timed) systems (see Section 2.1), on the other hand, have no distributed clock tree, they are event-driven. Their computations are triggered upon request, and they happen at their fastest possible speed in the specified operation and variation space. Moreover, there are no additional frequency adjustments, as required by the clocked approach. Self-timed logic provides better timing robustness under a variable voltage (Figure. 1.3 ("Design with timing assumptions" and "Delay insensitive design" areas)). An example of such a relationship can be found in Figure. 1.2, where Design 1 uses a speed-independent approach for a circuit which is built from dual-rail components with completion detection. On the one hand this design is more robust to delay variations due to low voltage supply levels, on the other it requires more power due to its additional circuitry. Design 2 employs a bundled-data approach, thus it is less timing robust on low power levels but has much less overhead for a nominal Vdd. In the

light of the above, the best way to implement a system that is both power-proportional and power-efficient in a wide range of supply voltage levels is to produce a hybrid design, as discussed earlier.

Let us now consider another example with two system design approaches, shown in Figure 1.4: a traditional clock-driven design and a self-timed energy-modulated system.

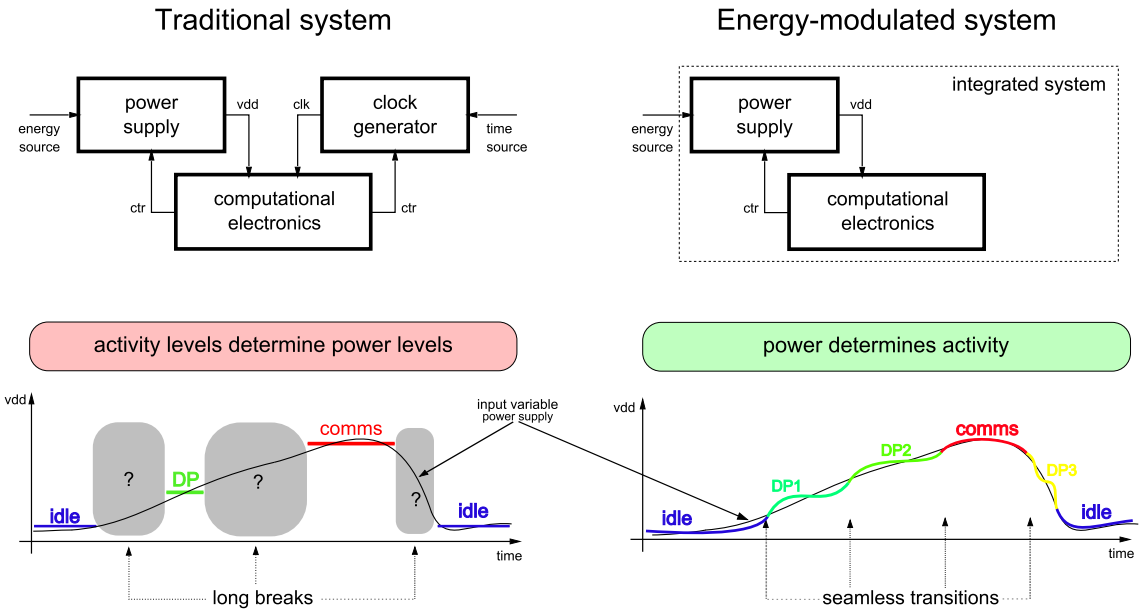


Figure 1.4: Traditional and energy-modulated system view.

Both of the examples contain several operating modes, i.e. idle mode, data processing (DP, DP1, DP2, etc.), communicating (comms), etc., each of which becomes active and performs to a certain level of quality in response to some level of power supply. In traditional (synchronous) systems the range of operating voltages for a specific mode is narrow. Therefore in a variable voltage supply we could have long breaks before a particular mode can be used. However in the asynchronous example this operating voltage range is much wider, hence these modes can start operating much earlier, and therefore we have a seamless transition between different modes as the power supply changes.

In light of the above, asynchronous methods are more suitable for design of power-proportional systems and can help to address the grand challenge of developing and producing asynchronous power-proportional designs.

The next section outlines research goals and overall contribution of the thesis.

1.3 Research contribution

The first stage of the research investigated the various models and formalisms (PN, FSMs, etc.) for control logic synthesis, as well as different methodologies for asynchronous systems design. Particular attention was paid to the Conditional Partial Order Graph (CPOG) methodology. Its feasibility was later demonstrated by an implementation of an asynchronous microcontroller [132].

As the research was progressing, the challenge became not only to implement a more sophisticated example, using the CPOG formalism, but to develop a system which will be able to meet a variety of functional requirements as well as be adjustable to the wide range of operating modes. In other words, it should be reconfigurable and power-proportional.

To demonstrate these ideas the Intel 8051 microprocessor was chosen. Its architecture and implementation are reasonably old (1980s), however there are still plenty of devices that use this CPU [7].

The main objective was to show the feasibility of our approach on a realistic microprocessor architecture, so that it could be further applied to the design of modern CPU architectures. In this aspect there are other interesting CPU architectures could be used as a a vehicle for this work, for instance MSP430 CPU from Texas Instruments [146]. On the one hand its original ISA contains only 27 instructions, which leads to a fairly simple control logic and won't show the capability and advantage of the CPOG method to work on a larger scale instruction set, as it was shown on a bigger ISA of the Intel 8051. On the other hand this simplicity of ISA can be used to expand the number of modes in which instructions can be used and therefore show the advantage of the CPOG approach to work with multi-modal systems.

The following outlines the most important contributions in this thesis:

- **Propose a design flow for the development of instruction set architectures for a microprocessor, which can be altered to suit a particular hardware platform or a**

particular operating mode.

Following the discussions in the introduction, we developed an Instruction Set Architecture (ISA) design flow. This flow uses a convenient and powerful formalism for specification of processor instruction sets called the CPOG model.

- **Development of an adaptive and reconfigurable system, based on an asynchronous Intel 8051 microprocessor, with run-time adaptability of its functionality and operation modes.**

  We implemented an asynchronous Intel 8051 CPU to demonstrate the feasibility of the CPOG formalism and the proposed ISA design flow in the development of a sophisticated microprocessor.

- **Testing of the adaptive design by implementing a proof-of-concept ASIC and evaluating its performance and power consumption.**

  The proposed reconfigurable design was implemented as a proof-of-concept ASIC. The chip went through a series of tests and evaluation stages. Measured results proved the feasibility of the proposed design flow and demonstrated the advantages of the adaptive design.

This work has been conducted as part of the PowerProp project funded by EPSRC EFuturesXD. The main goal of this project was to address a wide development gap between the ways of how energy efficiency is approached by hardware and software engineers. Therefore this work required active involvement of researchers from both the microelectronics and software engineering domains of Newcastle University. The project has also been influenced by an intern-ship at Imagination Technologies during the winter of 2012, where it was possible to experience the industrial aspects of hardware development and fabrication.

The work conducted to the above research goal resulted in a number of publications:

1. M. Rykunov, A. Mokhov, A. Yakovlev, A. Koelmans, “Specification and synthesis of processors using CPOG-based methodology”. Proceedings of the UK Electronics Forum. Newcastle, UK, 2010.

2. M. Rykunov, A. Mokhov, A. Yakovlev, A. Koelmans, "Automated Generation of Control logic for Processor Architectures". Proceedings of the UK Electronics Forum. Manchester, UK, 2011.

3. A. Mokhov, M. Rykunov, D. Sokolov, A. Yakovlev, "Formal modelling and transformations of processor instruction sets". Proceedings of the 9th ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE). Cambridge, UK, 2011.

4. M. Rykunov, A. Mokhov, D. Sokolov, A. Yakovlev, A. Koelmans, "Reconfiguration Strategies for Hardware-Software Energy Awareness". Proceedings of the UK Electronics Forum. Newcastle, UK, 2012.

5. M. Rykunov, A. Mokhov, A. Yakovlev, A. Koelmans, "Automated generation of processor architectures in embedded systems design". Technical Report NCL-EECE-MSD-TR-2010-164.; 2012.

6. A. Mokhov M. Rykunov D. Sokolov A. Yakovlev, A. Iliasov and A. Romanovsky, "Synthesis of processor instruction sets from high-level ISA specifications". IEEE Transactions on Computers, 2013.

7. M. Rykunov, A. Mokhov, D. Sokolov, A. Yakovlev and A. Koelmans, "Design-for-Adaptivity of Microarchitectures". Proceedings of the 24th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP13), Washington D.C., USA, 2013.

8. A. Mokhov, M. Rykunov, D. Sokolov, A. Yakovlev, "Towards Reconfigurable Processors for Power-Proportional Computing". Proceedings of the 12th IEEE Low Voltage Low Power Conference (FTFC). Paris, France, 2013.

1.4 Organisation of the thesis

The rest of the thesis is organised as follows:

Chapter 2 (Background) outlines the main classes of asynchronous circuits, gives an introduction into the CPOG model, talks about the main features of Intel 8051 microprocessor, its original architecture and an overview of various asynchronous implementations and finally discusses the main aspects of power proportional computing techniques.

Chapter 3 (The Design of Instruction Set Architecture) outlines the main aspects of designing instruction sets and presents a case study to show all benefits of the introduced compositional approach.

Chapter 4 (Design of Asynchronous 8051 Microprocessor) describes the main stages of the microprocessor design flow and provides implementation details for our asynchronous 8051 microprocessor.

Chapter 5 (Application example : Implementation of the demonstration chip) addresses the development of the asynchronous 8051 microprocessor and its ASIC implementation, which covers synthesis, verification, fabrication and testing.

Chapter 6 (Conclusions) summarises the contribution of the thesis and outlines areas of future work.

Appendix describes PO specifications of instructions in both control logics (the Top-level and the ALU); shows Boolean equations from the mapping of the CPOGs; provides more details on the bonding diagram of the chip.

Chapter 2

Background

This chapter gives the main background information for this thesis. The following areas are covered:

- **Asynchronous systems**

  Section 2.1 introduces basic properties and classes of self-timed designs and explains the main types of communication protocols used in asynchronous circuits.

- **Conditional Partial Order Graph methodology**

  In Section 2.2 we outline some essential information about a novel compositional approach based on Conditional Partial Order Graphs.

- **Intel 8051 microprocessor design and its asynchronous derivatives**

  The architecture and the main features of the Intel 8051 core are explained in Section 2.3. Along with the original design we give a quick overview of its main asynchronous variations.

- **Power-proportional computing**

  In Section 2.2 we overview the main techniques currently used in the area of power-proportional computing.

2.1 Asynchronous systems

Digital circuits register computation results when an operation completion signal is issued. In synchronous circuits the role of such a signal belongs to a global clock whose period is chosen to be long enough for all the circuit modules to complete the computation, thus exhibiting the worst case performance. Self-timed (or asynchronous) circuits have no distributed clock tree, hence the completion detection is achieved by requesting each module to indicate its progress independently, either through explicit completion detection logic or by replicating the critical path in the form of a matching delay line [139]. This approach gives the following advantages, which contribute to the popularity of the asynchronous design:

- **Low power consumption.** In a clocked design the distributed clock tree consumes nearly half of the total power consumption [54]. It still dissipates dynamic power even if there are no computations happening, just because it is clocked. There is no clock in asynchronous designs, therefore no meaningless switching activity (everything is event-driven) and no losses in power.

- **Potentially higher performance.** The maximum frequency of a clocked design is determined by the global worst case latency, hence each next computational cycle needs to wait for this delay. Self-timed systems are event-driven, i.e. the next computational cycle starts immediately after the previous one has indicated its completion. It should be mentioned that there are several techniques developed (see Section 2.1.2) in order to detect the completion the previous computation stage.

- **Low electromagnetic emission**. The absence of a clock in self-timed systems leads to lower levels of electromagnetic emission compared to the synchronous design. This property can be used for security applications as in a clocked design it is easier to extract information from it by using the clock patterns in its electromagnetic noise profile as a reference for the data flow.

- **Robustness towards process and supply voltages variations**. The global worst case latency in a synchronous design is determined for a very narrow safety mar-

gin on the process variation and operating conditions (i.e. voltage supply and temperature), hence any significant change of these parameters may cause a malfunctioning of the design. Self-timed circuits adjust to these variation naturally, if the computation time of any of the units changes (due to these variations) the performance of the whole system changes accordingly.

Despite these advantages the design and synthesis of asynchronous circuits are still more of an academic exercise than the mainstream of the semiconductor industry. This is due to significant changes required in the conventional design flow, the immaturity of the software tools and long learning curve for engineers. The most successful commercial solution is provided by Handshake Solutions [69] in their Timeless Design Environment (TiDE). Its open-source alternative is the Balsa toolset [2]. Other examples of asynchronous design flows are BESST [21], TAST [144], PipeFitter [120], etc.

2.1.1 Classes of asynchronous circuits

There are several classes of asynchronous circuit, that one can distinguish. The *Delay Insensitive* (DI) class is the most robust to process and environmental variations [41]. The DI approach makes no assumptions on wire or gate delays, therefore such circuits can correctly operate with the unbounded gate and wire delay models (the formal specification can be found in [149]). Despite these advantages very few examples of this class can be found in real life due to significant difficulties of its implementation using standard logic gates as well as performance and area overheads [94].

In order to be able to build practical circuits out of standard gates it is necessary to loosen the DI restrictions, hence the *Speed-independent* (SI) approach was developed. Similar to the DI approach, SI circuits assume the unbounded gate delay model, however the delay of wires is considered to be "negligible", so that the output of one gate is immediately propagated to another gate [111]. In 1960x the first asynchronous SI microprocessor (ILLIAC II) was developed, which was the most powerful computer at the time [112].

The *Quasi Delay Insensitive* (QDI) class assumes a "negligible" wire delay, as in SI

circuits, introduces the concept of the isochronic fork. This fork is a wire fanout, which has matching signal transitions at all ends of the fork. In other words it is assumed that there is no delays between the branches of wire fork [93].

There are numerous applications of QDI and SI circuits, some examples of which are outlined in Section 2.3.

The presented approaches are classic ways for designing a self-timed circuit from "scratch". However the re-design of existing IP cores in an asynchronous style is not acceptable for industry due to time to market constraints. Recently a less intrusive desynchronisation technique found its way to commercial products [82]. It converts synchronous circuits into asynchronous ones at a late stage of the conventional design flow, thus reusing time-proved synchronous EDA tools. There are several ways to ensure asynchronous operation of the resultant circuits, e.g. in the Nanochronous [114] implementation a copy of the circuit critical path is used to adapt the clocking speed to the environment variations.

2.1.2 Datapath encoding schemes

There are two main commonly used datapath encodings used for implementing asynchronous circuits: *dual-rail* and *bundled-data* protocols.

The bundled-data protocol represents each bit of data by one single wire. Request and acknowledgement signals are separate and bundled with the data. Two signalling disciplines can be exercised over a bundled-data channel – 2-phase and 4-phase. A 2-phase protocol indicates the availability of results by any change of the completion signal. However in a 4-phase protocol the completion signal needs to return to zero, representing the mandatory reset stage, before starting the next round of handshaking operation. The 2-phase protocol is potentially faster than the 4-phase since there is a latency overhead because of the mandatory reset phase, however the implementation of it is much more complex and hard to design, therefore it often results in a large overhead in terms of area and power consumption [139].

Opposite to the bundled-data approach the dual-rail protocol uses two wires to re-

Table 2.1: Dual-rail data encoding

State	data true	data false	Description
Empty	0	0	Reset or spacer state is used after the recipient acknowledged the data to separate two valid data sets
Valid "1"	1	0	Logic "1"
Valid "0"	0	1	Logic "0"
Not valid	1	1	Not used

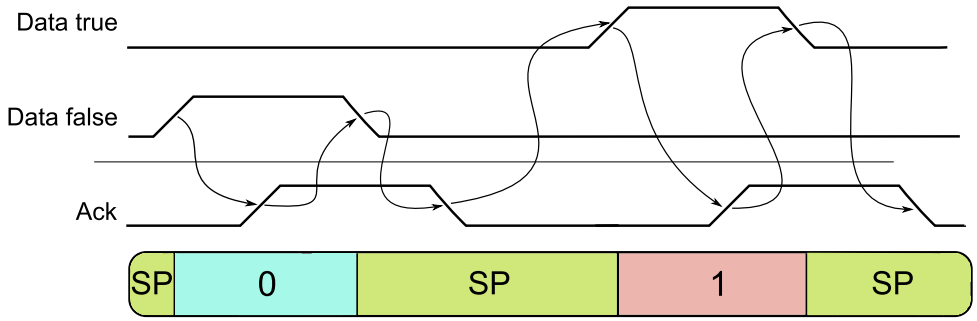


Figure 2.1: Dual-rail protocol

present one bit of data and one separate handshake signal, representing request and acknowledgement. One of the data wires is called "data true" and the other – "data false". Both of them are used in a specific dual-rail data encoding (see Table 2.1 and Figure 2.1).

Similar to the the bundled-data protocol, dual-rail can also be separated in to 4-phase and 2-phase approaches. It is mandatory that in the 4-phase approach there should be a spacer in-between two valid data symbols, however in the 2-phase approach each valid data symbols comes immediately after another one has been acknowledged.

Comparing the bundled-data and the dual-rail protocols one can notice their respective advantages and disadvantages. The bundled-data protocol needs only a single wire to represent one bit of data, this simplifies the datapath logic. This leads to smaller circuits, which consume much less power, compared to the dual-rail protocol. Generally with proper delay matching, bundled-data circuits operate faster than the dual-rail protocol especially with a large data bus, as the performance doesn't suffer from the complicated completion detection circuitry as in the the dual-rail protocol.

In light of the above the bundled-data 4-phase protocol was chosen for our design (see Section 4.1).

2.2 Essentials of Conditional Partial Order Graph formalism

Conditional Partial Order Graphs (CPOGs) are a novel compositional approach, which is capable of capturing similar behavioural patterns, or event orders, in a compact functional form. In particular this approach is beneficial for systems with many behavioural scenarios defined on the same set of primitive actions, e.g. CPU microcontrollers. Using this approach the whole microcontroller's design flow becomes highly efficient as it is based only on structural methods and does not require exploration of the entire controller state space or explicit enumeration of all its behavioural scenarios.

This section will focus on the essential information on the CPOG methodology, that originally was developed on the basis of well-studied and closely related Partial Orders (POs) and Directed Acyclic Graphs (DAGs) formalisms [23, 49, 90].

2.2.1 Essentials of CPOGs

A *Conditional Partial Order Graph* [107] (further referred to as *CPOG* or simply *graph*) is a quintuple $H = (V, E, X, \rho, \phi)$ where:

- V is a set of *vertices* which correspond to events (or atomic actions) in a modelled system.
- $E \subseteq V \times V$ is a set of *arcs* representing dependencies between the events.
- *Operational vector* X is a set of Boolean variables. An *opcode* is an assignment $(x_1, x_2, \ldots, x_{|X|}) \in \{0,1\}^{|X|}$ of these variables. An opcode selects a particular partial order from those contained in the graph.
- $\rho \in \mathcal{F}(X)$ is a *restriction function,* where $\mathcal{F}(X)$ is the set of all Boolean functions over variables in X. ρ defines the *operational domain* of the graph: X can be assigned only those opcodes $(x_1, x_2, \ldots, x_{|X|})$ which satisfy the restriction function, i.e. $\rho(x_1, x_2, \ldots, x_{|X|}) = 1$.

(a) Full notation (b) Simplified notation

Figure 2.2: Graphical representation of CPOGs

- Function $\phi : (V \cup E) \to \mathcal{F}(X)$ assigns a Boolean *condition* $\phi(z) \in \mathcal{F}(X)$ to every vertex and arc $z \in V \cup E$ in the graph. Let us also define $\phi(z) \stackrel{\text{df}}{=} 0$ for $z \notin V \cup E$ for convenience.

CPOGs are represented graphically by drawing a labelled circle ○ for every vertex and drawing a labelled arrow ⟶ for every arc. The label of a vertex v consists of the vertex name, a colon and the vertex condition $\phi(v)$, while every arc e is labelled with the corresponding arc condition $\phi(e)$. The restriction function ρ is depicted in a box next to the graph; operational variables X can therefore be observed as parameters of ρ.

Fig. 2.2(a) shows an example of a CPOG with $|V| = 5$ vertices and $|E| = 7$ arcs. There is a single operational variable x; the restriction function is $\rho(x) = 1$, hence both opcodes $x = 0$ and $x = 1$ are allowed. Vertices $\{a, b, d\}$ have constant $\phi = 1$ conditions and are called *unconditional*, while vertices $\{c, e\}$ are *conditional* and have conditions $\phi(c) = x$ and $\phi(e) = \overline{x}$ respectively. Arcs also fall into two classes: *unconditional* (arc $c \to d$) and *conditional* (all the rest). As CPOGs tend to have many unconditional vertices and arcs we use a simplified notation in which conditions equal to 1 are not depicted in the graph; see Fig. 2.2(b).

The purpose of conditions ϕ is to 'switch off' some vertices and/or arcs in a CPOG according to a given opcode, thereby producing different *CPOG projections*. An example of a graph and its two projections is presented in Fig. 2.3. The leftmost projection is obtained by keeping in the graph only those vertices and arcs whose conditions evaluate to 1 after substitution of variable x with 1 (such projections are conventionally denoted by $H|_{x=1}$). Hence, vertex e disappears (shown as a dashed circle ◌), because its condition

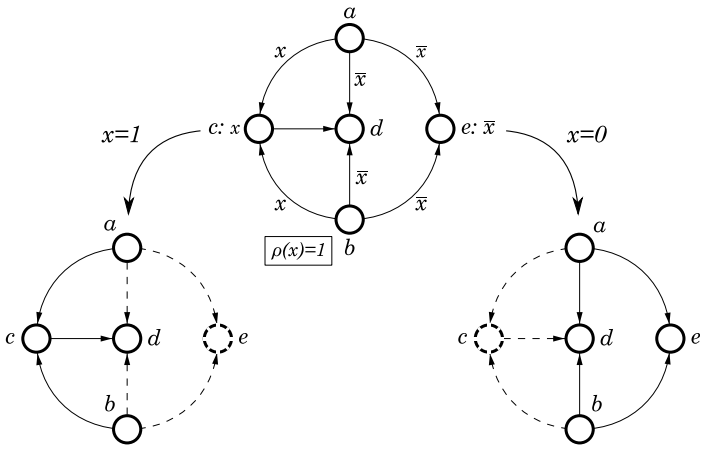


Figure 2.3: CPOG projections: $H|_{x=1}$ (left) and $H|_{x=0}$ (right)

evaluates to 0: $\phi(e) = \overline{x} = \overline{1} = 0$. Arcs $\{a \to d, a \to e, b \to d, b \to e\}$ disappear for the same reason; they are shown as dashed arrows ⇢. The rightmost projection is obtained in the same way with the only difference that variable x is set to 0; it is denoted by $H|_{x=0}$, respectively. Note that although the condition of arc $c \to d$ evaluates to 1 (in fact it is constant 1) the arc is still excluded from the resultant graph because one of the vertices it connects, viz. vertex c, is excluded and naturally an arc cannot appear in a graph without one of its vertices. Each of the obtained projections can be regarded as the specification of a particular behavioural scenario of the modelled system, e.g. as specification of a processor instruction. Potentially, a CPOG $H = (V, E, X, \rho, \phi)$ can specify an exponential number of different instructions (each composed from atomic actions in V) according to one of $2^{|X|}$ different possible opcodes.

2.3 Intel 8051 Microcontroller

In this Section we focus on the main features of the Intel 8051 microprocessor and its original architecture (Sections 2.3.2 and 2.3.3), and give a quick overview of various asynchronous implementations (Section 2.3.4).

2.3.1 Introduction

The original synchronous 8051 microcontroller (MCU) was developed by Intel in the early 1980s using NMOS technology. In later versions, it was moved to the CMOS, hence the name was changed to 80C51. Up to the present time the Intel 80C51 microcontroller and its numerous derivatives are widely used all over the globe; 8051 is one of the most widely produced 8-bit microcontroller in the world [92]. Nearly every major semiconductor manufacturer, such as Infineon, Philips, Atmel, STMicroelectronics, Texas Instruments, etc., has their own version of the 8051. Usually all the 8051 derivitives are based on the same CPU architecture, but differ in sizes and types of the memories, and in the peripherals.

In general a microcontroller is a "small computer" situated on a single IC consisting of two main parts: the CPU core with its memories and input/output peripheral blocks (e.g timers, counters, receivers and transmitters, etc. [158]). The main target in our work was designing a CPU, therefore we concentrated on the processor of the Intel 8051 microcontroller. The next two subsections will address the architecture, the main features and the ISA of this core.

2.3.2 Intel 80C51 microprocessor core

The original CPU core adopts the Harvard architecture [45], which separates the storage of data and instructions.

The data memory usually is split into internal (256 byte) and external (64 kbyte) Random Access Memory (RAM) blocks. The internal data RAM contains several dedicated areas: the first 64 bytes (00h - 1Fh) are four *register banks* of eight registers each; then a small memory area (20h - 2Fh) is bit-addressable space; the next 80 bytes (30h - 7Fh) are usually shared between the stack data and user variables; finally the last 128 bytes are a special part of the internal RAM known as the space of the *Special Function Registers* (SFRs). These are readable and writeable registers (such as accumulator, B-register, Program State Word (PSW), Data PoinTeR (DPTR), etc.) could be accessed by the CPU. The handbook of the standard 8051 shows that the SFR-space is not completely filled

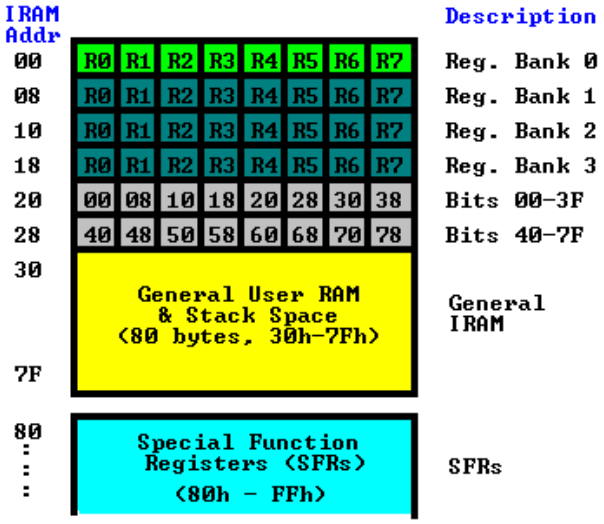


Figure 2.4: Organisation of the internal memory in the Intel 8051 microprocessor

by these special registers, which only use 21 out of 128 spaces, the rest is free for user variables. The structure of the RAM block is shown in Figure 2.4 [1] or can also be found in the original 8051 handbook [158].

The program memory is most commonly implemented as an off-chip EPROM, so that it's more convenient for reprogramming.

The synchronous architecture of the Intel 8051 is shown in Figure 2.5 [8]. The core is built around the *internal bus* (IB), to which all main registers are able to write to and to read from. We can see all the main components, such as the Arithmetic Logic Unit (ALU), Read Only Memory (ROM), Random Access Memory (RAM), the SFR-space and the four bidirectional ports to the outside world. There is a special separate *B* bus, which is used for modifying the program counter (PC). Having only two main buses to communicate between the registers makes this architecture very compact and efficient. However this significantly reduces parallelism in the system, hence all the instructions contain many sequential parts in their execution.

2.3.3 Instruction Set and addressing modes

ISA is usually divided into the following classes: Complex Instruction Set Computer (CISC), Reduced Instruction Set Computing (RISC), Very-Long Instruction Word (VLIW)

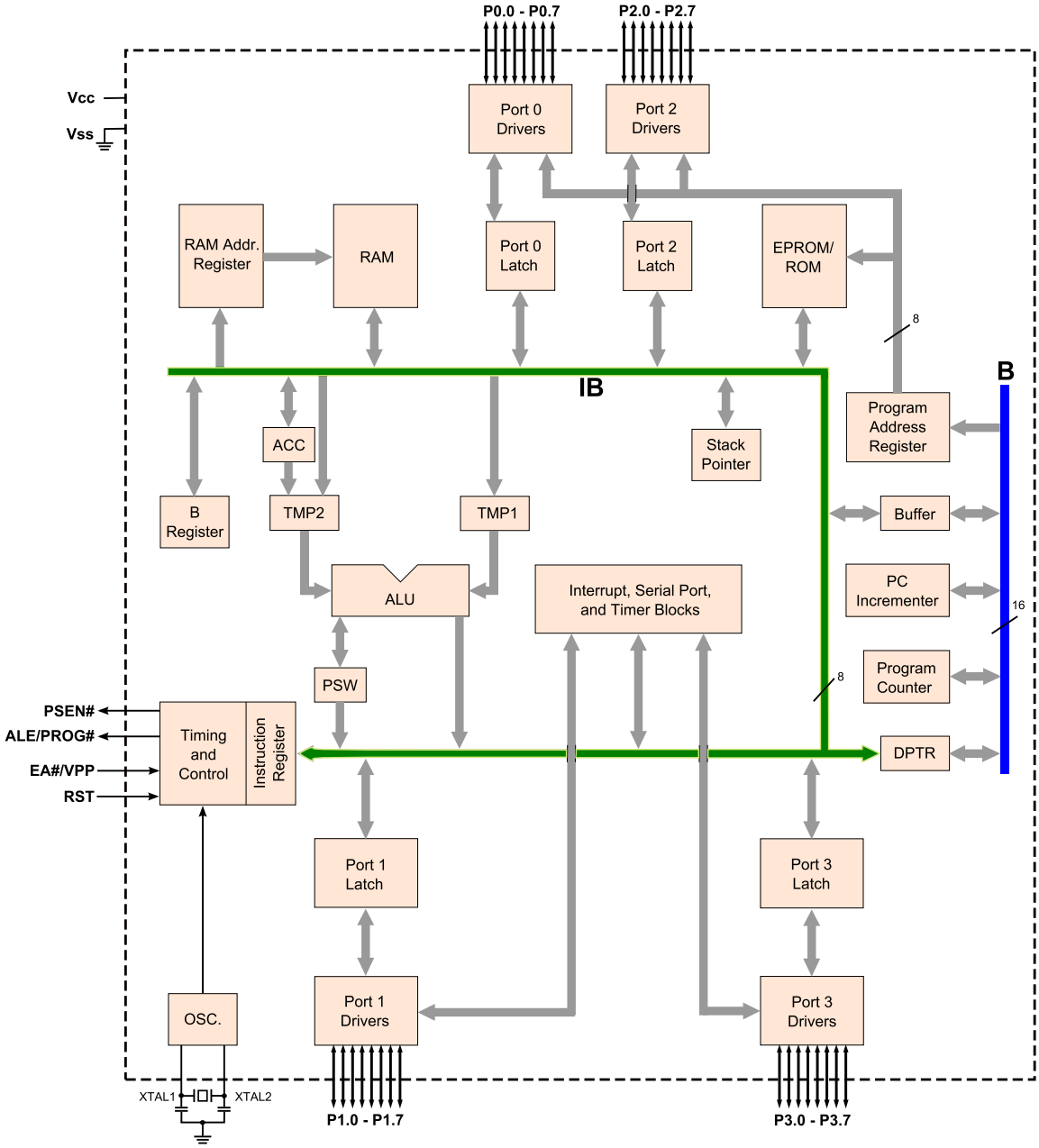


Figure 2.5: Architecture of a synchronous Intel 8051 microprocessor

and various hybrids [62, 34]. These classes differ in the complexity of the their instruction sets, the encoding style of the instructions, and the homogeneity of the internal register structure in the implementation.

Due to the following facts the 8051 microprocessor can be considered as a CISC architecture:

- The instruction set has 255 different complex instruction with variable length (from 1 to 3 bytes), many of which involve multiple memory accesses.
- A number of different addressing modes, such as immediate, register addressing, direct, indirect, relative and indexed, which are explained later on.
- The system doesn't have uniform internal register structure. There are several specialised registers (SFRs, four register banks, etc.).
- It takes a variable number of clock cycles to execute an instruction. Each instruction takes one, two or four *machine cycles* to execute. Each machine cycle consists of six *slots*, each of which performs different operations and requires one clock cycle [158].

In contrast to CISC, RISC and VLIW architectures usually have fixed length instructions with a regular format. Usually it is a one simple (RISC) or many independent simple (VLIW) operations. The structure of the register file is regular with many general-purpose registers. The hardware design usually focuses on designing high-performance implementations, where the execution of several instructions can be pipelined, as opposed to CISC, which exploits microcoded implementations, i.e multiple operations are encoded in one instruction.

The instruction set can be separated into five classes:

- **Arithmetic operations**: this class of operations includes addition (ADD), addition with carry (ADDC), subtraction (SUBB), increment (INC), decrement (DEC), multiplication (MUL), division (DIV) and decimal adjustment of accumulator (DA).
- **Logic operations**: this class of operations includes logic AND (ANL), logic OR (ORL), logic exclusive OR (XRL), clear (CLR), complement (CPL), swap (SWAP), rotate to

the right (RR), rotate to the right with carry (RRC), rotate to the left (RL) and rotate to the left with carry (RLC).

- **Data transfer operations**: moving data from and to internal (MOV) and external (MOVX, MOVC) data memory, push data to (PUSH) and from (POP) the stack and the exchange operation (XCH).
- **Boolean operations**: these instructions operate on individual bits of registers.
- **Branching operations**: this class of instruction can conditionally and unconditionally change the contents of the PC. There are three main types: the short jump (SJMP), the long jump (LJMP) and the absolute jump (AJMP).

The full explanation of each instruction and their opcodes can be found in the Appendix.

Six addressing modes are supported by the 8051 instruction set:

- **Direct addressing mode**: the operand is specified by an 8-bit address field. This addressing mode is used only for accessing the data in the internal RAM. For example, DEC 01h (operation: (R1):= (R1) – 1, 01h is the direct address of the second register in the first bank).
- **Immediate addressing mode** – the value is a constant and, as the name suggests, it is stored immediately after the operation code in memory. For example, ADD A, #123h (operation: (A):= (A) + 123h).
- **Register addressing mode** involves the use of the Bank of registers to hold the data to be manipulated. The 3-bit register specification is part of the opcode of the instruction. For example, INC R6 (operation: (R6):= (R6) + 1).
- **Indirect addressing mode** is used when the instruction performs an operation on the data whose address is contained in register R0 or R1. For example, DEC @R0 (decrement the internal RAM cell by 1 indirectly through the R0 register).
- **Relative addressing mode** is used with jump instructions, when a fetched address is loaded to the PC. For example, SJMP #11h (operation: (PC):= (PC) + 11h).

- **Indexed addressing mode** is used for accessing data elements of look-up table entries in the ROM, where the address of the data in the table is formed by adding the accumulator and base pointer. For example, MOVC A, @A+DPTR (move the code data relative to the DPTR to the accumulator (address=A+DPTR)r).

More specific and detailed information on the 8051 architecture can be found in numerous publications and user manuals [158, 80, 48].

2.3.4 Overview of Asynchronous Intel 8051 implementations

In the introduction we mentioned that there are numerous derivatives of the synchronous implementation of the Intel 8051 microprocessor. However many designers also focused on its asynchronous implementation.

Probably one of the first asynchronous 8051 microcontroller was implemented by Gageldonk et al. [151] in the Philips Research Laboratories in 1998 and later became a commercial product. In fact this MCU was developed using the Tangram (or Haste) behaviour model, which originally was introduced by the Philips Research Laboratories in the Tangram tool over 20 years ago [150]. Later on this work proceeded by Handshake Solutions [69] and led two implementations HT80C51-LP (Low Power) and HT80C51-LC (Low Cost). Both of them were mainly designed to demonstrate the feasibility of the Tangram design flow.

In 2002 Lee et al. proposed a new version of a self-timed 8051 with a new 5-staged pipelined architecture [81]. This implementation regrouped the entire ISA into seven groups, which were defined by a particular execution scheme of the instructions. In this way some instructions needed only parts of the scheme, e.g,. NOP instruction needed only fetching and decoding, and therefore regrouped instructions use the same set of pipelined stages.

A year later a QDI asynchronous 8051 microcontroller called Lutonium [96] was introduced. This design utilised highly parallel processing with a deep pipeline architecture. The main core was described with Communicating Hardware Process (CHP) [95] and the pipeline stages were implemented using the Pre-Charge Half Buffers (PCHB) template.

There were also several other implementations using CSP-like hardware description languages such as the open-source Balsa toolset developed at Manchester University [2, 17], which led to RTL simulations and an FPGA implementation. This was a 2-stage pipeline architecture, which used a partial instruction decoding [35]. Another example is a pipelined asynchronous 8051 soft-core , which was implemented and validated on an FPGA [36].

One of the most recent implementations of this core was done by Chang et al. [88]. In this work two 8051 microcontroller cores were designed on the same die: one synchronous and one asynchronous (QDI approach). The main target of this work was to delineate and compare these two approaches with the same environment and variation conditions, as they were physically located on the same chip.

Each of these asynchronous 8051 microcontroller examples was done in a different process size and a different purpose was targeted, therefore we didn't compare them in terms of power and performance. However we did compare our implementation with some of these approaches in the Implementation chapter (Chapter 5).

2.4 Power-proportional computing

In the introduction we discussed the main aspects of power-proportionality and how one can introduce them in the system development flow, particularly in asynchronous systems (see Section 1.2). In this section we show the main techniques and approaches people use in their circuit development process nowadays.

At the present time such methods have been partially addressed within the already well-established research area called low-power IC design in the form of fairly special techniques for reducing the switching activity (dynamic power) and leakage current (static power) in the circuit.

In the beginning of the CMOS technology era dynamic power has been dominating in logic. In synchronous circuits up to 50% of dynamic power goes to global clock distribution across the chip. This became the primary target for power saving in *clock gating* techniques, where clock switching is suppressed for inactive parts of the system. There

is a trade-off between the granularity of clock gating and the area overheads introduced by gating logic. All modern synthesis tools support basic RTL-level clock gating, while tools developed in Calypto [31] and Envis [56] extend this approach by comprehensive analysis of the circuit to provide optimal clock gating solution.

In deep sub-micron technology the trend has changed and static power is no longer negligible - up to 40% of the total power is due to leakage. This is usually resolved by *power gating*, where the voltage source is disconnected from those parts of the circuit which are inactive for extended periods of time. Over-conservative variation margins on the clock period are utilised in the *voltage scaling* approach, which is a more aggressive technique for dynamic power reduction [162]. There are several approaches to voltage scaling:

- Static Voltage Scaling (SVS): different blocks or subsystems are given carefully selected fixed supply voltages.
- Multi-level Voltage Scaling (MVS): an extension of static voltage scaling where a block or a subsystem is switched between two or more voltage levels (independent power supplies). Only a few statically selected levels are supported for different operating modes.
- Dynamic Voltage and Frequency Scaling (DVFS): an extension of MVS where a larger number of voltage levels are dynamically switched between to follow changing workloads.
- Adaptive Voltage Scaling (AVS): an extension of DVFS where a control loop is used to adjust the voltage. This approach is implemented using off-chip voltage regulators by National Powerwise [115] in a publicly available tool [83].

There are several low-level techniques for decreasing the leakage of the cells outside the speed-critical path either by using a special low-leakage technology library (supported by all modern synthesis tools and many libraries have low-leakage gate implementations) or by adjusting their lithography mask data (e.g. implemented in Blaze DFM tools, Tela Innovations company [145] and used at the TSMC foundry).

Significant improvements can be achieved by a more radical approach - conversion of circuits to an asynchronous mode of operation. Self-timed circuits are free from a rigid clock and function at the best speed for given operating conditions, where e.g. voltage scaling fits naturally, which we demonstrated in this work.

Chapter 3

The design of Instruction Set Architecture

The design of a microprocessor or any other complicated circuitry is not a trivial process, and consists of a deep analysis beforehand and various implementation stages afterwards [97]. First and foremost we need to think what functions, architecture and structure we want to have in the future design. In terms of the microprocessor's development, its structure and functionality highly depends on the Instruction Set Architecture (ISA), which is used during its design flow.

Optimal design of an instruction set for a particular combination of available hardware resources and software requirements is crucial for building processors with high performance and energy efficiency, and is a challenging task involving a lot of heuristics and high-level design decisions.

Design of the microprocessor ISA is a computationally intensive task whose search space grows exponentially with the number of instructions and supported operating modes. Furthermore, the ISA development process often goes beyond a one-time effort of a single designer as the ISA may need to be extended at the customer side, e.g., as in Application Specific Instruction set Processors (ASIPs) [152]. ASIPs allow adding new functionality to an extensible baseline ISA in the form of Instruction Set Extensions (ISEs), thereby combining the flexibility of a general purpose CPU and performance of an ASIC.

The key idea is to analyse the application domain and identify repetitive source code fragments that can be replaced by custom ISE instructions to reduce overheads associated with the instruction fetch cycle and storage of temporary values [68], as well as to enable additional optimisation opportunities in resource allocation, register binding, and port assignment [40][121].

Modern embedded systems often require yet another dimension of ISA flexibility – dynamic reconfigurability. For example, a baseband processor whose core functionality is signal processing may need to be reconfigured upon standardisation of a new communication protocol. Reconfigurable ASIPs address this requirement by combining a static general purpose ISA with a reconfigurable fabric to introduce new functionality when it becomes needed [26][27]. Reconfigurability and custom instructions also address the issue of energy efficiency (a major concern for the microelectronics industry, particularly in mobile and embedded domains) by power elasticity [161] and by moving computationally intensive algorithms from software to hardware [68][89].

One of the key difficulties in designing instruction sets is the necessity to comprehend and deal with a large number of instructions, whose microcontrol implementation may be altered to suit a particular hardware platform or a particular operating mode (Section 3.1). To overcome this, instructions and groups of instructions have to be managed in a compositional way: an ISA specification should be composable from specifications of its constituent parts (Section 3.2). Furthermore, one should be able to transform and optimise ISA specifications (Section 3.3) in a fully formal way to guarantee correctness without computationally expensive verification after each incremental modification of an ISA (Section 3.4). The chapter is concluded with a case study in Section 3.5 to demonstrate how CPOGs can be used for capturing different hardware configurations and operation modes.

3.1 Which ISA to choose?

As it was mentioned in the beginning of the Chapter, the design of a microprocessor's ISA is one of the main parts in the design flow of a processor. There are several criteria which

determine the choice of an instruction set for a needed processor microarchitecture.

Functionality. Each instruction is associated with a sequence of atomic *actions* (usually acyclic) to complete the corresponding computational task. Note that while a sequential run of actions is sufficient to achieve the instruction functionality, it is often practical to enable some of the actions concurrently, e.g., in order to speed up the instruction execution and to efficiently utilise the available energy. The distinctive classes of instruction functionality are arithmetic operations, data handling, memory access and flow control.

The amount of computation per instruction is an important characteristic of an ISA, which can be illustrated by comparing CISC, RISC and VLIW architectures (see Section 2.3.3). The CISC architecture is based on a semantically rich instruction set, which provides operand access in several addressing modes and can execute complex multi-cycle operations without storing the intermediate results [72]. In contrast, the RISC architecture employs a relatively small set of basic instructions to build a complex functionality at the level of software [43]. The microarchitecture complexity of the VLIW architecture falls between the RISC and CISC architectures, as the scheduling for Instruction Level Parallelism (ILP) is performed statically during the program compilation, when VLIW instruction is broken into several simple RISC instructions [59].

Operation modes. The same functionality can be achieved in different ways targeting various optimisation criteria. For example, an arithmetic operation can be executed either in an energy efficient way but slowly, or in a low latency mode at the price of extra energy consumption. Alternatively, for security applications, the operation can be combined with power masking and data scrambling. The choice of available operation modes is usually made at the design time and is limited by the circuit area and the timing constraints. Selection of the operation mode can be encoded in the instruction set at two levels: *coarse-grain*, as a separate class of mode-switching instructions or *fine-grain*, as a part of each instruction code.

For example, in the ARM architecture [61], apart from the standard RISC-like operation mode with a 32-bit instruction set there are several special modes, e.g., Thumb and

Jazelle. In the Thumb mode the processor switches to a compact 16-bit encoding of a subset of ARM instructions and makes the instruction operands implicit. This reduces the processor functionality but improves its power efficiency through increased code density, usually at the expense of performance. In the Jazelle mode the instruction set is changed to natively execute Java Bytecode and to support just-in-time compilation [113].

Resources. At least one functional unit must be available for each type of atomic action comprising the instructions. The conflicting situations, when the same hardware resource is requested by several actions, are resolved through scheduling and may also involve dynamic arbitration. The quantity of each resource type is therefore decided by trading resource idle time against the frequency of potential conflicts to resolve.

Modern CPUs, while often referred to as RISC-like, also exhibit the features of CISC and VLIW architectures. For example, they often have complex multi-clock DSP/multi-media instructions, which is typical for CISC. They also combine the compile-time VLIW scheduling with dynamic arbitration of resources to employ ILP for instruction pipelining, out-of-order and speculative execution. Such a diversity of instruction functionality, combined with various operation modes and resource constraints, makes ISA design extremely challenging.

3.1.1 Existing ISA approaches and challenges

There are several well-established approaches for the functional-level description and formal verification of an ISA. *Event-B* [164] is a widely adopted language for specifying *first-order logic* systems and doing refinements on these representations. Combined with the Rodin theorem prover [148], it becomes a powerful platform for proving that a (refined) system satisfies the initial specification, e.g., does not leave a certain set of 'good' states during its operation. *HOL* [60] is a computer-assisted proving environment for constructing verifiably correct mathematical proofs. Although its expressiveness is unrivalled, the generic nature of a tool such as Isabelle/HOL makes it more suitable for analysing individual instructions with deep mathematical properties; see, for example, verification of the IA-64 division algorithm [70].

These formal ISA methods have a history of being used for reasoning about hardware implementations, however they are more targeted at the software-related aspects of processor functionality. No hardware implementation issues are usually taken into consideration apart from those directly visible to the instructions, such as the size of addressable memory, the number and type of available registers, etc. As a result, an ISA designer does not have the full control on how the specified functionality is achieved in hardware, what the costs of every instruction are in terms of energy consumption and computation resources, how to minimise latency of instruction decoding logic, or how to dynamically adapt the processor to the current operating conditions. Modelling such low-level implementation details in Event-B or HOL is costly; a more targeted formalism is needed to interface the representation of knowledge about instructions sets with that of knowledge about their execution.

There is clearly a niche in microprocessor EDA where the following design requirements need to be addressed:

- description of individual instruction functionalities at the microcode level as partial orders of atomic actions;
- efficient representation and manipulation with complete instruction sets (re-encoding, re-targeting, etc.);
- compositional approach to ISA design to facilitate modularity, extensibility and reuse;
- explicit capturing of processor operation modes;
- possibility to express the resource availability constraints.

We propose to address these requirements using the Conditional Partial Order Graphs (CPOGs) approach [107]. This model is particularly convenient for composition and representation of large sets of partial orders in a compact form. It can be equipped with a suite of mathematical tools for the refinement, optimisation, encoding and synthesis of the control hardware which implements the required instruction set, similar in spirit to the approach based on *control automata* [15]. We envisage that the model can be

used as a complementary formalism for the existing ISA methodologies providing a formal link between the software and hardware domains. Although general-purpose modelling languages and proving environments, such as Event-B or HOL, may be used to a similar effect, the CPOG model offers a superior mathematical construction permitting automated analysis and synthesis.

Moreover the area of ASIP also contributes by this approach by providing a methodology to systematically manipulate instruction sets in order to explore the space of possible solutions. Our approach can simplify the design of ASIPs and synthesis of ISEs, as it naturally supports incremental and compositional development of instruction sets. Moreover, we utilise the same formal model throughout the whole design process: specification of individual instructions, combining them into instruction sets, exploring the design space, and synthesis of the control logic [106], which facilitates productivity and consistency of the design flow.

Figure 3.1 shows the proposed pathway from a high-level specification of an ISA to a low-level microcontroller implementation. Our specification and synthesis flow comprises four distinct levels. At the *architectural level* the ISA is modelled using the Event-B formalism. Given available hardware resources and operating modes we can refine the ISA and descend to the *microarchitectural level*. At the *transformation level* the refined instructions are composed into a single CPOG representation which is then iteratively optimised for a set of design constraints, such as requirements to the instruction opcodes and ILP support. Finally, at the *implementation level* the ISA is synthesised into a set of hardware components, such as instruction decoder and microcontrol logic.

In the next Section we describe the use of the CPOG method for the specification of processor instruction sets and demonstrate the approach on an example.

3.2 Specification of instructions CPOG model

The essentials of the Conditional Partial Order Graph methodology was presented in Section 2.2, therefore in this section we discuss the formal correspondence between a CPOG representation and CPU instruction (see Section 3.2.1). Also we discus how this

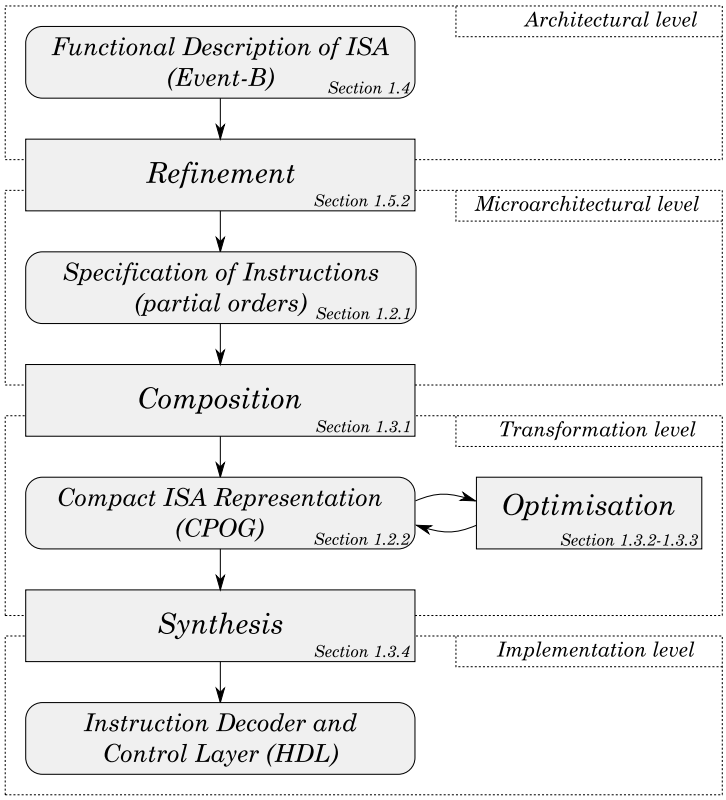


Figure 3.1: Specification and synthesis flow

specification method can be expanded to the whole ISA.

3.2.1 Specification of instructions

Consider a processing unit that has two registers A and B, and can perform two different instructions: *addition* and *exchange* of two variables stored in memory. The processor contains five datapath components (denoted by $a \ldots e$) that can perform the following atomic actions:

a) Load register A from memory;

b) Load register B from memory;

c) Compute sum $A + B$ and store it in A;

d) Save register A into memory;

e) Save register B into memory.

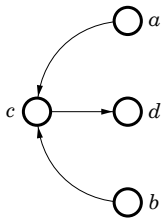
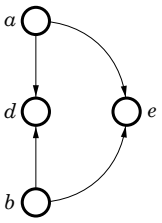
Instruction	Addition	Exchange
Action sequence	a) Load A b) Load B c) Add B to A d) Save A	a) Load A b) Load B d) Save A e) Save B
Partial order with maximum concurrency	 P_{ADD}	 P_{XCHG}

Table 3.1: Two instructions specified as partial orders

Table 3.1 describes the addition and exchange instructions in terms of usage of these atomic actions.

The addition instruction consists of loading the two operands from memory (actions a and b, causally independent and thus possibly concurrent), their addition (action c), and saving the result (action d). Whether a and b are to be performed concurrently depends on: i) the system architecture, e.g., if concurrent read memory access is allowed, ii) static and dynamic resources availability (the processor hardware configuration must physically contain two memory access components and they both have to be immediately available for use), and iii) the current operation mode which determines the scheduling strategy, e.g. 'execute a and b concurrently to minimise latency', or 'execute a and b in sequence to reduce peak power'. Let us assume for simplicity that in this example all causally independent actions are always performed concurrently, see the corresponding partial order P_{ADD} in Table 3.1[1]. Section 3.5 will address joint specification of different scheduling strategies of an instruction.

The operation of exchange consists of loading the operands (concurrent actions a and b), and saving them into swapped memory locations (concurrent actions d and e), as captured by P_{XCHG}. Note that in order to start saving one of the registers it is necessary

[1]In this example we describe partial orders using *Hasse diagrams* [23], i.e. without depicting transitive dependencies, such as, for example, dependencies $a \rightarrow d$ and $b \rightarrow d$ in partial order P_{ADD}.

to wait until both of them have been loaded to avoid overwriting one of the values.

One can see that the two partial orders in Table 3.1 appear to be the two projections shown in Figure 2.3, thus the corresponding graph can be considered as a joint specification of both instructions. Two important characteristics of such a specification are that the common events $\{a, b, d\}$ are overlaid and the choice between the two operations is distributed in the Boolean expressions associated with the vertices and arcs of the graph. As a result, in our model there is no need for a 'nodal point' of choice, which tend to appear in alternative specification models (a Petri Net [50] would have an explicit choice place, a Finite State Machine [100] – an explicit choice state, and a specification written in a Hardware Description Language [100] would describe the two instructions by two separate branches of a conditional statement `if` or `case`). The absence of a choice nodal point could lead to a confusion, as this point would be "distributed" and won't be clearly seen on a PO representation. Usually such a nodal point gives us a condition for a choice, however in CPOG representation only by applying a particular condition we can see different brunches.

One downside of a purely graph-based approach to instruction sets is the inability to reason about functional correctness; specifically, the relationship between an instruction behaviour and the functionality of the blocks it is made of. Clearly, a designer would seek some form of assurance that an instruction is correct in respect to original requirements and an evidence of correctness is exhibited. An ultimate form of evidence is a formal proof. In Section 3.4 we will show how to obtain the proof of instruction correctness with a refinement-based derivation of instruction logic.

3.2.2 From instructions to instruction sets

The following notions are introduced to formally define specification and composition of instruction sets.

An *instruction* is a pair $I = (\psi, P)$, where $\psi \in \{0,1\}^{|X|}$ is a vector assigning a Boolean value to each variable in X, and $P = (V, \prec)$ is a partial order defined on a set of atomic

actions V. Semantically, ψ represents the instruction opcode[2], while the precedence relation $\prec$ of the partial order captures the behaviour of the instruction[3]. We assume that V and X belong to the corresponding universes shared by all the instructions of the processor: $V \subseteq \mathcal{U}_V$ and $X \subseteq \mathcal{U}_X$.

An *instruction set* (denoted by IS) is a set of instructions with unique opcodes, i.e. for any $IS = \{I_1, I_2, \ldots, I_n\}$, such that $I_k = (\psi_k, P_k)$, all opcodes ψ_k must be different.

Given a CPOG $H = (V, E, X, \rho, \phi)$ there is a natural correspondence between its projections and instructions: an opcode $\psi = (x_1, x_2, \ldots, x_{|X|})$ induces a partial order $H|_\psi$, and paired together they form an instruction $I_\psi = (\psi, H|_\psi)$ according to the above definition. This leads to the following formal link between CPOGs and instruction sets.

A CPOG $H = (V, E, X, \rho, \phi)$ is a *specification* of an instruction set $IS(H)$ defined as a union of instructions $(\psi, H|_\psi)$ which are allowed by the restriction function ρ (see Section 2.2):

$$IS(H) \stackrel{\text{df}}{=} \{(\psi, H|_\psi),\ \rho(\psi) = 1\}. \tag{3.1}$$

Using this definition we can formally state that the graph in Figure 2.3 specifies the instruction set from Table 3.1. Section 3.3 shows how to obtain and efficiently manipulate such CPOG specifications.

3.3 Transformations

In this section we describe CPOG transformations which allow the systematic manipulate of instruction sets. The transformations facilitate the following stages of the ISA design flow shown in Figure 3.1:

- *compositional and modular construction* of instruction sets from smaller subsets and/or individual instructions (Section 3.3.1);
- *global ISA modifications*, that is modifications of all the instructions at once, for

[2] In this section the instruction operands are implicit and the opcode completely defines the instruction. We elaborate on this in Section 3.5.

[3] We incorporate the notion of a *microprogram* [100] (the behaviour of the instruction) into the definition of the instruction.

example, re-encoding, re-targeting for a different hardware platform, refinement for hardware synthesis (Section 3.3.2);

- *local and incremental ISA modifications*, which usually apply only to a subset of all the instructions and are heavily relied on in various ISA optimisation algorithms (Section 3.3.3);

- *hardware synthesis*, i.e., transformation of an instruction set into a microcontroller by mapping a given CPOG into Boolean equations (Section 3.3.4).

An important feature of all the discussed transformation procedures is their higher efficiency in comparison to the conventional approaches. In particular, we will demonstrate that the algorithmic complexity of all the procedures does not depend on the number of instructions in a given ISA.

3.3.1 Composition

Compositionality is a key concept in modern system design: a realistic system can only be designed and analysed by breaking it down into smaller pieces. A typical instruction set of a modern processor contains hundreds of base instruction classes and various ISA extensions, and usually is a result of several design iterations. Therefore, it is necessary to be able to compose large instruction sets from smaller ones to enable modularisation, reuse, and incremental development.

A CPOG can be deconstructed by means of projections, as was demonstrated in Figure 2.3. The opposite operation, that is constructing a CPOG out of given parts, is called *composition*. This subsection describes how it can be used to build large instruction sets from smaller ones.

Definition 3.1. Two well-formed graphs H_1 and H_2 are said to be in an *encoding conflict* with respect to their restriction functions ρ_1 and ρ_2 iff $\rho_1\rho_2 \neq 0$. An encoding conflict implies the existence of an opcode ψ such that both of the restriction functions are satisfied: $\rho_1|_\psi = \rho_2|_\psi = 1$. This leads to ambiguity in some cases, when two graphs describe different behaviour for the same opcode ψ. Depending on whether these two

graphs actually specify the same or different scenarios under ψ the conflict can be either true or false.

An encoding conflict is *true* if the partial orders generated with ψ are different:

$$\exists\psi,\ (\rho_1\rho_2)|_\psi = 1,\ \mathbf{po}(\mathbf{dg}\ H_1|_\psi) \neq \mathbf{po}(\mathbf{dg}\ H_2|_\psi)$$

Conversely, an encoding conflict is *false* if the partial orders generated with ψ are in fact the same:

$$\forall\psi,\ (\rho_1\rho_2)|_\psi = 1,\ \mathbf{po}(\mathbf{dg}\ H_1|_\psi) = \mathbf{po}(\mathbf{dg}\ H_2|_\psi)$$

Formally, the *composition* of two instruction sets IS_1 and IS_2 is simply defined as their union $IS_1 \cup IS_2$; it is required that the union does not contain two instructions with the same opcode otherwise it would be impossible to distinguished them during the decoding process. Due to the commutativity and associativity properties of set union $\cup$, one can compose more than two instruction sets by performing their pairwise composition in arbitrary order, for instance, $IS_1 \cup IS_2 \cup IS_3 = (IS_1 \cup IS_2) \cup IS_3 = IS_1 \cup (IS_2 \cup IS_3)$.

Note that if instructions in given sets IS_k are represented individually (e.g., by listing them one after another as in conventional methods), then the complexity of the composition operation is linear with respect to the total number of instructions: $\Theta(|IS|)$, where $IS = \bigcup_k IS_k$. This is because we have to iterate over all of them to generate the result. It may be unacceptably slow for those applications which routinely perform various operations on large instruction sets. By using the CPOG model for the compact representation of instruction sets, one can perform most of the operations much faster, as demonstrated below.

Let instruction sets IS_1 and IS_2 be specified with graphs $H_1 = (V_1, E_1, X, \rho_1, \phi_1)$ and $H_2 = (V_2, E_2, X, \rho_2, \phi_2)$, respectively, as in (3.1), where the set of variables X is the same. Then their composition has CPOG specification $H = (V_1 \cup V_2, E_1 \cup E_2, X, \rho_1 + \rho_2, \phi)$, where the vertex/arc conditions ϕ are defined as

$$\forall z \in V_1 \cup V_2 \cup E_1 \cup E_2,\ \phi(z) \stackrel{\text{df}}{=} \rho_1\phi_1(z) + \rho_2\phi_2(z).$$

We call H the *CPOG composition* of H_1 and H_2 and denote this operation as $H = H_1 \cup H_2$. Note that if $\rho_1 \cdot \rho_2 \neq 0$ then the composition is undefined, because $IS(H_1)$ and $IS(H_2)$ contain instructions with the same opcode ψ allowed by both restriction functions: $\rho_1(\psi) = \rho_2(\psi) = 1$. The case of graph addition was introduced previously [102]. The following theorems highlight the key properties of the composition operation regarding the union of graphs.

Theorem 3.1. *Union $\cup$ is an associative and commutative operation, when its arguments are not in conflict.*

Proof. 1) Associativity: $(H_1 \cup H_2) \cup H_3 = H_1 \cup (H_2 \cup H_3)$.

Follows from the associativity of set union ($(V_1 \cup V_2) \cup V_3 = V_1 \cup (V_2 \cup V_3)$ etc.) and Boolean disjunction ($(\rho_1 + \rho_2) + \rho_3 = \rho_1 + (\rho_2 + \rho_3)$). To prove associativity with respect to conditions ϕ, let us define ρ' and ϕ' to be the restriction functions and conditions of graph $H' = H_1 \cup H_2$: $\rho' = \rho_1 + \rho_2$ and $\phi' = \rho_1\phi_1 + \rho_2\phi_2$. In the same way, let ρ and ϕ denote the restriction function and conditions of the final graph $H = H' \cup H_3$. So, $\rho = \rho' + \rho_3 = \rho_1 + \rho_2 + \rho_3$ while ϕ is equal to

The result remains the same if the order of union of the three graphs is altered: $H' = H_2 + H_3$, $H = H_1 + H'$. So, independently of the order, function $\phi(z)$ for a particular z will eventually be equal to $\rho_1\phi_1(z) + \rho_2\phi_2(z) + \rho_3\phi_3(z)$.

2) Commutativity: $H_1 \cup H_2 = H_2 \cup H_1$.

Follows from the commutativity of set union ($V_1 \cup V_2 = V_2 \cup V_1$ etc.) and Boolean disjunction ($\rho_1 + \rho_2 = \rho_2 + \rho_1$ etc.) operations. □

Remark 3.1. When more than two graphs are in union then the redundant brackets can be omitted without ambiguity: $H_1 \cup H_2 \cup H_3$.

Corollary 1. *The general equation for conditions ϕ in graph $H(V, E, X, \rho, \phi)$ in case of union of $n \geqslant 2$ graphs $H_k(V_k, E_k, X, \rho_k, \phi_k)$, $1 \leqslant k \leqslant n$ is*

$$\phi = \sum_{1 \leqslant k \leqslant n} \rho_k \phi_k$$

e.g. if $n = 3$ the equation is $\phi = \rho_1\phi_1 + \rho_2\phi_2 + \rho_3\phi_3$.

$$\phi = \rho'\phi' + \rho_3\phi_3 =$$

$$= (\rho_1 + \rho_2)(\rho_1\phi_1 + \rho_2\phi_2) + \rho_3\phi_3 =$$

$$= \rho_1\rho_1\phi_1 + \rho_1\rho_2\phi_2 + \rho_2\rho_1\phi_1 + \rho_2\rho_2\phi_2 + \rho_3\phi_3 =$$

Since H_1 and H_2 are not in conflict, then $\rho_1 \cdot \rho_2 = 0$

$$= \rho_1\phi_1 + 0\phi_2 + 0\phi_1 + \rho_2\phi_2 + \rho_3\phi_3 =$$

$$= \rho_1\phi_1 + \rho_2\phi_2 + \rho_3\phi_3$$

Theorem 3.2. *If* H_1 *and* H_2 *are not in conflict then*

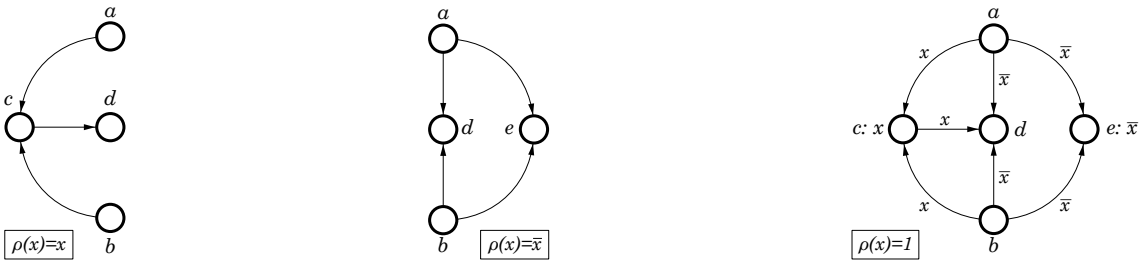
$$IS(H_1 \cup H_2) = IS(H_1) \cup IS(H_2)$$

i.e. graph $H_1 \cup H_2$ *contains partial orders from both* H_1 *and* H_2.[4]

Proof. Let $H = H_1 \cup H_2$. At first let us show that $IS(H_1) \cup IS(H_2) \subseteq IS(H)$. Consider an instruction (see definition in Section 3.2.2) $I \in IS(H_1)$ (the proof for the case when $I \in IS(H_2)$ is similar due to symmetry between H_1 and H_2).

1. The restriction function $\rho_2|_\psi$ of H_2 is not satisfied because H_1 and H_2 are not in conflict: $(\rho_1\rho_2)|_\psi = \rho_1|_\psi \cdot \rho_2|_\psi = 1 \cdot \rho_2|_\psi = \rho_2|_\psi = 0$.
2. The restriction function ρ of H is satisfied with ψ: $\rho|_\psi = (\rho_1 + \rho_2)|_\psi = 1 + 0 = 1$.
3. Vertex/arc conditions $\phi(z)$ for $\forall z \in V_1 \cup E_1$ in $H|_\psi$ evaluate to the same values as in $H_1|_\psi$: $\phi(z)|_\psi = (\rho_1\phi_1(z) + \rho_2\phi_2(z))|_\psi = 1 \cdot \phi_1(z)|_\psi + 0 \cdot \phi_2(z)|_\psi = \phi_1(z)|_\psi$.

[4]Moreover union preserves the initial opcodes of the partial orders.



(a) H_{ADD} (b) H_{XCHG} (c) $H_{ADD} \cup H_{XCHG}$

Figure 3.2: Graph composition

Therefore, the sets of vertices and arcs of $H|_\psi$ are the same as those of $H_1|_\psi$. Consequently, $P = H_1|_\psi = H|_\psi$ and therefore $I = (\psi, P) \in IS(H)$.

Now let us prove the reverse statement: $IS(H) \subseteq IS(H_1) \cup IS(H_2)$. Consider an instruction $I = (\psi, P) \in IS(H)$. The restriction function $\rho = \rho_1 + \rho_2$ must be satisfied which means that either ρ_1 or ρ_2 is satisfied but not both of them. Let it be ρ_1: $\rho_1|_\psi = 1$ and $\rho_2|_\psi = 0$ (the other case is again symmetric). This leads to the same conclusion as in the first part of the proof : $H_1|_\psi = H|_\psi$. Therefore $IS(H) \subseteq IS(H_1) \cup IS(H_2)$. This completes the proof. □

Crucially, the complexity of computing a CPOG composition does not depend on the total number of instructions $|IS_1 \cup IS_2|$. It depends only on the sizes of graph specifications H_1 and H_2: $\Theta(|V_1|+|E_1|+|V_2|+|E_2|)$. Since the number of arcs $|E_k|$ is at most quadratic with respect to $|V_k|$ and $|V_k| \leqslant |U_V|$ (all vertices are contained in universe U_V), we have the following upper bound on CPOG composition complexity: $O(|U_V|^2)$. Note that $|U_V|^2$ is potentially smaller than the number of different instructions[5], which can be exponential with respect to $|V|$, in particular the total number of partial orders on set U_V is greater than $2^{\frac{1}{4}|U_V|^2}$ [23]. To conclude, we can operate on the CPOG representations of instruction sets faster than on the instruction sets themselves.

Let us demonstrate the composition of instruction sets on the aforementioned processing unit example. Figure 3.2(a,b) shows two graphs H_{ADD} and H_{XCHG} specifying

[5]Although this statement does not hold for our simplistic examples, e.g., $|V|+|E| = 5+7 = 12$ and $|IS| = 2$ in Figure 3.2, it does hold in practice. For example, our implementation of Intel 8051 microprocessor (see Section 4.2 has 257 instructions but its CPOG representation contains only 17 vertices and 46 arcs. Moreover, if we do not use abstraction and treat instructions *ADD A,B* and *ADD C,D* as different ones, the number of instructions of a modern 32-bit processor can easily grow to 2^{32} while its CPOG will remain compact.

singleton instruction sets $IS(H_{ADD}) = \{(1, P_{ADD})\}$ and $IS(H_{XCHG}) = \{(0, P_{XCHG})\}$, respectively. Since their restriction functions are orthogonal $\rho_{ADD} \cdot \rho_{XCHG} = x \cdot \overline{x} = 0$, we can compose them into the graph shown in Figure 3.2(c). It specifies the composition $IS(H_{ADD} \cup H_{XCHG}) = \{(1, P_{ADD}), (0, P_{XCHG})\}$ as intended (see Figure 2.3).

3.3.2 Global transformations

Consider a graph $H = (V, E, X, \rho, \phi)$. Since elements of the quintuple are shared by all instructions in $IS(H)$, we can make global modifications of the instruction set without iterating over all the instructions. For example, we can add a new action *go* at the beginning of every instruction by setting $V' = V \cup \{go\}$, $\phi(go) = 1$, and $\phi(go \to v) = 1$ for all $v \in V$. The cost of this global modification is only $\Theta(|V|)$; we call transformations of this type *vertex insertions*.

It is possible to introduce a global *concurrency reduction* between actions a and b, by setting $E' = E \cup \{a \to b\}$ and $\phi(a \to b) = 1$. As a result, action b will always be scheduled after a in *all* the instructions. The cost of this transformation is $O(1)$, but it is not safe in general: it can introduce deadlocks if action a is scheduled to happen after b in one of the instructions (forming a cyclic dependency). To ensure deadlock freeness verification algorithms from [102] must be employed.

Variable substitution is another basic transformation with the global effect. For instance, by replacing every occurrence of x with $\overline{x}$ in all conditions ϕ and function ρ, we flip the corresponding bit in all instruction opcodes. To perform this operation we need to change $\Theta(|V|^2)$ Boolean functions. Variable substitution is a powerful transformation, it can affect not only a single bit, but all the opcodes; care must be taken to ensure that the resultant opcodes do not clash and become in conflict.

Variable substitution is also applied to simplify the calculation, e.g. this technique is used a lot in Integration by substitution simplification.

A global *Opcode expansion* is used, when we want to introduce a new variable in the opcode and therefore all the conditions in the graph need to be changed. Let's assume we have two instructions, which are distinguished by one condition y, so the set of conditions

would be $X = \{y\}$. Now we want to add a third instruction, so we need extend the set of variables by a new condition x and new set will be $X' = X \cup \{x\}$. To perform this operation we need to change $\Theta(|V|^2)$ Boolean functions.

Opposite to the previous one, we can think of *Opcode reduction*. This transformation is performed when we want to optimise one of the variables away, so a new set of conditions would be $X' = X \setminus \{x\}$. With the same difficulty of $\Theta(|V|^2)$ we need to go through all the conditions, however we need to make sure that the rest of the opcodes do not in conflict.

The above transformations are *global*. It is possible, however, to apply them only to a subset of selected instructions using the operations of *set extraction* and *decomposition* defined below.

3.3.3 Local transformations

Instead of looking at the whole instruction set of a processor one may need to focus attention on its smaller parts. As an example, consider the *MMIX processor* instruction set [86] containing 256 different opcodes. 16 of them, starting with bits 0010, are dedicated to addition/subtraction operations, and a designer wants to manipulate them separately from the others.

Let graph $H = (V, E, X, \rho, \phi)$ specify the whole instruction set $IS(H)$ of the processor and 8-bit opcodes be encoded with variables $\{x_1, \ldots, x_8\}$. Function $f = \overline{x_1} \cdot \overline{x_2} \cdot x_3 \cdot \overline{x_4}$ enumerates all Boolean vectors starting with 0010 and its conjunction with ρ enumerates all wanted opcodes. Thus, graph $H' = (V, E, X, f \cdot \rho, \phi)$ specifies the required part of $IS(H)$. There is a dedicated operation in the CPOG algebra, called *scalar multiplication*, specifically intended for this task: $H' = f \cdot H$ [107]. Its main feature is that

$$\forall f,\ IS(f \cdot H) \subseteq IS(H)$$

In our context, f can be considered an *instruction property* and operation $f \cdot H$ can be called a *set extraction*: it extracts a subset of a given instruction set according to a required property.

A generalisation of this operation is called *decomposition*. It is easy to see that

$H_1 = f \cdot H$ and $H_0 = \bar{f} \cdot H$ together contain all instructions from $IS(H)$: the instructions with opcodes satisfying property f are put into H_1, and all the rest are put into H_0. Thus, any instruction set can be decomposed into two disjoint sets according to a given property. This is formally captured by the following statement:

$$\forall f,\ IS(H) = IS(f \cdot H) \cup IS(\bar{f} \cdot H)$$

Set extraction and decomposition are cheap operations: they only require computation of a conjunction of two Boolean functions f and ρ.

Returning back to the MMIX example, we can decompose $IS(H)$ into two disjoint sets: addition/subtraction operations $IS_1 = IS(f \cdot H)$, and all the rest $IS_0 = IS(\bar{f} \cdot H)$. Then we can apply a transformation, e.g., an event insertion, to IS_1 resulting in IS_1^t. Finally, we can compute composition $IS^t = IS_1^t \cup IS_0$ which contains all the instructions from the original instruction set $IS(H)$, but with a *local* transformation applied only to addition/subtraction operations.

3.3.4 Mapping to logic gates

Finally, the refined CPOG can be mapped into Boolean equations and produce a physical implementation of the specified microcontroller. In order to descend from the abstract level of atomic actions to the physical level of digital circuits signal-level refinements are necessary.

To interface with an asynchronous datapath component a it is possible to use the standard request-acknowledgement handshake (req_a, ack_a), as shown in Figure 3.3. In case of a synchronous component b the request signal is used to start the computation but, as there is no completion detection, the acknowledgement signal has to be generated using a matched delay [139]. Also, there are cases when a matched delay has to be replaced with a counter connected to the *clock* signal to provide an accurate multi-cycle delay – see the interface of component c in the same figure. Note that we do not explicitly show *synchronisers* [85] in the diagram; it is assumed that components b and c are equipped with the necessary synchronisation mechanisms to accept asynchronous

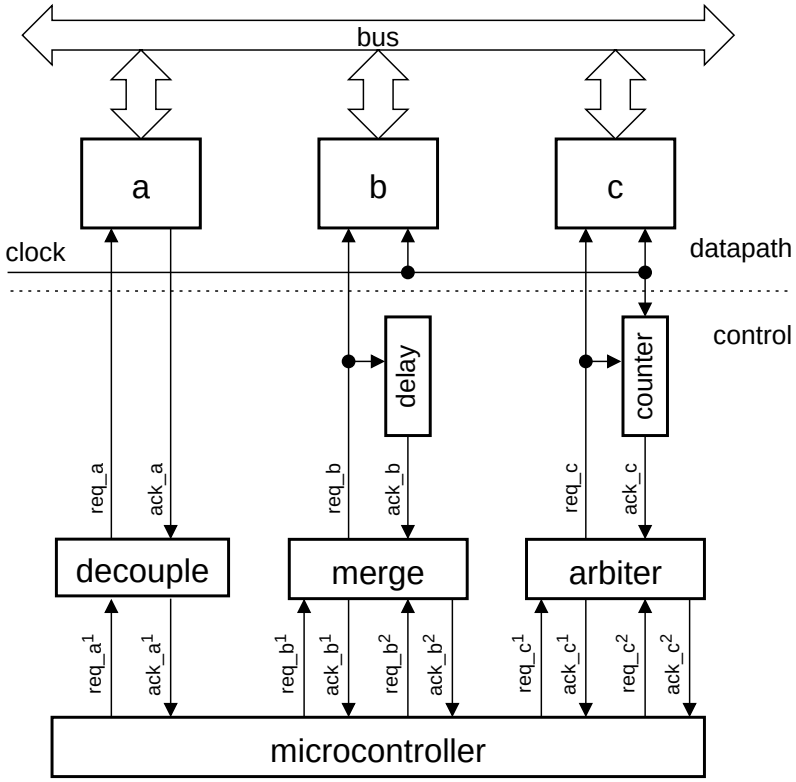


Figure 3.3: Datapath interface architecture

requests from the microcontroller.

To explicitly specify handshake signals it is possible to perform a graph transformation explained in Figure 3.4. Every atomic action a^1 is split into a pair of events req_a^1+ and ack_a^1+ standing for rising transitions of the corresponding handshake signals. If there are two occurrences of an atomic action, e.g. b^1 and b^2, then both vertices are split[6], etc. Semantically, when an atomic action a^1 is ready for execution, the controller should issue the request signal req_a^1 to component a; then the high value of the acknowledgement signal ack_a^1 will indicate the completion of a.

Notice that the microcontroller does not reset handshakes until all of them are complete. This leads to a potential problem: a component cannot be released until the instruction execution is finished. To deal with the problem it is necessary to *decouple* the microcontroller from the component, see box 'decouple' in Figure 3.3 and its gate-level implementation in Figure 3.5(a). Also, when a component b is used twice in an instruction we have to combine two handshakes $(req\_b^{1,2}, ack\_b^{1,2})$ into one using the

[6]We use superscripts to distinguish different occurrences of the same event.

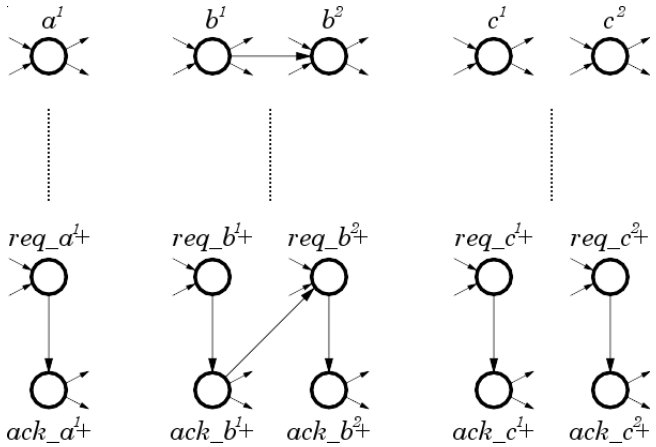


Figure 3.4: Signal-level refinement

merge controller, see Figure 3.5(b). Merge controllers can only be used if the requests are mutually exclusive[7]. If this is not the case, as e.g. for concurrent actions c^1 and c^2, then we have to set an *arbiter* guarding access to the component. Its implementation consists of the merge controller and the *mutual exclusion (ME) element* [85], see Figure 3.5(c).

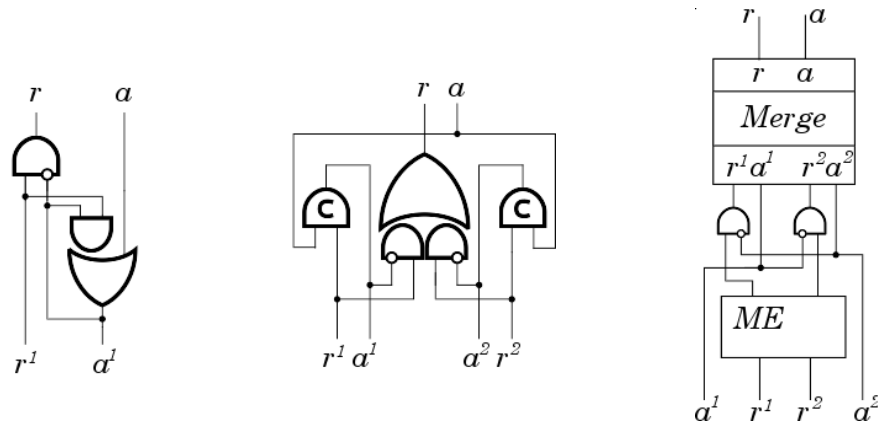
Finally, the refined graph can be mapped into Boolean equations. An event associated with vertex $v \in V$ is enabled to fire (req_v+ is excited) when all the preceding events $u \in V$ have already fired (ack_u have been received):

$$req\_v = \phi(v) \cdot \prod_{u \in V} \left(\phi(u) \cdot \phi(u \to v) \Rightarrow ack\_u\right)$$

where $a \Rightarrow b$ stands for Boolean implication indicating 'b if a' relation. Mapping is a simple structural operation, however the obtained equations may not be optimal and should undergo the conventional logic minimisation [100, 107] and technology mapping [50] procedures.

It is interesting to note that the size of the microcontroller does not depend on the number of instructions directly. There are $\Theta(|V|^2)$ conditions ϕ in all the resultant equations; the average size of these conditions is difficult to estimate, but in practice we found that the overall size of the microcontroller never grows beyond $\Theta(|V|^2)$.

[7] It is possible to formally verify if two events in a CPOG are mutually exclusive using CPOG verification techniques from [108].



(a) Decouple (b) Merge (c) Arbiter

Figure 3.5: Handshake controllers

3.4 Functional correctness

In this section we discuss a formalism called Event-B [11] and its application to formal verification of correctness of CPOG-based representations of instructions. Event-B belongs to a family of state-based modelling languages that represent a design as a combination of state (a vector of variables) and state transformations (computations updating variables). In general, a design in Event-B is abstract: it relies on data types and state transformations that are not directly realisable. This permits terse models abstracting away from insignificant details and enables one to capture various phenomena of a system with a varying degree of detail. Crucially, each statement about the effect of a certain computation is supported by a formal proof. In Event-B, one is able to make statements about safety (this incorporates the property of functional correctness) and progress. Safety properties ensure that a system never arrives at a state that is deemed unsafe (e.g., keep on using power hungry computing blocks when there is a lack of energy in the battery). Progress properties ensure that a system is able to achieve its operational goals.

3.4.1 General Event-B methodology

An Event-B development starts with the creation of an abstract specification. A cornerstone of the Event-B method is the stepwise development that facilitates a gradual design

of a system implementation through a number of correctness-preserving *refinement* steps. The general form of an Event-B model (or *machine*) is shown in Figure 3.6. Such a model encapsulates a local state (program variables) and provides operations on the state. The actions (called *events*) are defined by a list of new local variables (parameters) vl, a state predicate g called *event guard*, and a next-state relation S called *substitution* (see the EVENTS section in Figure 3.6).

The INVARIANT clause contains the properties of the system (expressed as state predicates) that should be preserved during system execution. These define *safe states* of a system. In order for a model to be consistent, invariant preservation should be formally demonstrated. Data types, constants and relevant axioms are defined in a separate component called *context*.

Model correctness is demonstrated by generating and discharging *proof obligations* – theorems in first order logic. The proof obligations demonstrate model consistency, such as the preservation of the invariant by the events, and refinement links to other Event-B models. A collection of automated theorem provers attempts to discharge proof obligations; typically only 3%-5% of proofs require user intervention.

If a model possesses rich control flow properties (e.g., a computational algorithm) the control flow aspect of a model is defined in a separate view called the flow of a model [78]. The flow aspects introduces further verification obligations to ensure that all specified event ordering are found among event traces of a specification. In this work we apply the flow aspect to obtain structured programs – programs that use concepts like sequential composition, choice and loop.

3.4.2 Modelling instructions

Our goal is the verification of an instruction, that is, explaining how it is assembled from smaller blocks and whether such an assembly always delivers the right results. Before one may attempt such a verification, it is requisite to obtain a formal specification of what an instruction is expected to do. In other words, what is the expected effect of an instruction execution on system memory, registers and flags. Such a specification must

```
MACHINE M
  SEES Context
  VARIABLES v
  INVARIANT I(c,s,v)
  INITIALISATION R(c,s,v')
  EVENTS
    E1  =  any vl where
             g(c,s,vl,v)
           then
             S(c,s,vl,v,v')
           end
    ...
END
```

Figure 3.6: Event-B model structure

capture both the normal and abnormal cases. A normal case is a successful execution of an instruction until the completion; this happens when an instruction is called in a right state and with appropriate parameters. For some instructions, there are side conditions that must be satisfied or an instruction execution is aborted. One may also want to foresee (and, possibly, try to mask) abortive execution attempts due to transient hardware faults.

For a refinement-based approach such as Event-B the conventional way to obtain a specification is to gradually develop it from a high-level abstraction of a computing platform: memory that may be written and read, and a device acting upon it [32, 46]. Several specifications have been developed recently, e.g., for XMOS architecture [165], that employ Event-B to formalise instruction sets of real-life CPUs. A CPU is treated as a black-box so that a specification ends with a characterisation of normal and abnormal instruction behaviours. We take such a specification as our starting point, open the black box and explain how each instruction is realised.

Let us first examine what constitutes an instruction specification (Figure 3.6). The relevant ingredients are state variables (capturing concepts like memory, stack and registers), invariant and the pre- and postconditions of normal and abnormal instruction cases. Model variables v abstractly characterise memory and CPU states. An invariant $I(v)$ defines a set of safe states $S = \{v \mid I(v)\}$ that includes all the reachable model states; it is guaranteed that no chain of instruction execution could lead to a state outside S. Predicate $R(c,s,v')$ defines the set of vectors of initial variable values.

Let predicate families $P^i_N(v)$ and $Q^i_N(v,v')$ denote pre- and postconditions of normal instruction cases, where v and v' correspond to the current and the next states. Correspondingly, $P^i_A(v)$ and $Q^i_A(v,v')$ define abnormal cases.

For instruction preconditions $P_N(v)$ and $P_A(v)$ it holds that whenever an instruction is invoked and the system is in a safe state the instruction is ready to run:

$$I(v) \Rightarrow \bigvee_i P^i_N(v) \vee \bigvee_i P^i_A(v).$$

At the same time, there must be a definite way to tell which case applies in a current state and there should not exist a state where both normal and abnormal cases may be executed. As the system has a deterministic behaviour it can only be either in normal or abnormal state not in both. Formally, the normal and abnormal preconditions of an instruction must partition the set S of safe states:

$$S = \{v \mid P_N(v)\} \oplus \{v \mid P_A(v)\}.$$

A postcondition expresses the set of states that may be reached via an instruction execution (an instruction specification may be non-deterministic) and the relationship to the original state. An instruction must terminate in a safe state; that is, re-establish the invariant condition $I(v)$:

$$\forall i, t \cdot t \in \{N, A\} \wedge I(v) \wedge P^i_t(v) \wedge Q^i_t(v,v') \Rightarrow I(v').$$

The condition may be satisfied by simply choosing a pair of $P^i_t(v)$ and $Q^i_t(v,v')$ such that the left-hand side is always false. To counteract this, it is required that an instruction is always able to deliver some result:

$$I(v) \wedge P^i_t(v) \Rightarrow \exists v' \cdot Q^i_t(v,v').$$

The condition also captures the cases where a contradiction is present only for a subset of states characterised by $I(v) \wedge P^i_t(v)$, e.g., a pair of predicates $(y > 0, y' * y' = y)$ where

$y \in \mathbb{N}$ do not define a valid instruction case.

In a general case, an instruction specification is formed of a number of normal and abnormal cases.

instruction name is

state v

invariant $I(v)$

behaviour

$P_N^1(v) \rightarrow Q_N^1(v, v')$

...

$P_N^k(v) \rightarrow Q_N^k(v, v')$

$P_A^1(v) \rightarrow Q_A^1(v, v')$

...

$P_A^k(v) \rightarrow Q_A^k(v, v')$

end

An instruction implementation explains how each case of an instruction specification is implemented by a deterministic program comprising of primitive functional blocks.

To formally relate an operation specification to an implementation we construct a separate Event-B development for each case of an operation. An abstract machine of such development is based on the following template.

MACHINE op

VARIABLES m, r, f, c

INVARIANT

$I_m(m, r, f)$

$c \in \mathbb{B}$

$c = \text{FALSE} \Rightarrow P(m, r, f)$

$c = \text{TRUE} \Rightarrow Q(m, r, f)$

INITIALISATION

$m, r, f, c :| I_m(m', r', f') \wedge P(m', r', f') \parallel c := \text{FALSE}$

EVENTS

op = when

$c = \text{FALSE}$

then

$m, r, f, c :| Q(m', r', f') \parallel c := \text{TRUE}$

end

END

Here, I_m is the state model of an instruction, c is an auxiliary control variable. The model defines a single step automata. The automata is initialised into a state when $c = \text{FALSE}$ and atomically transitions into a terminal state where $c = \text{TRUE}$. The invariant properties $c = \text{FALSE} \Rightarrow P(m, r, f)$ and $c = \text{TRUE} \Rightarrow Q(m, r, f)$ explain the meaning of the automata states in relation to the operation definitions: initially, the state satisfies the operation precondition; upon termination it satisfies the operation postcondition. A single transition, defined in event op, takes the automata from a state satisfying the precondition to a state satisfying the postcondition. Thus, the specification is trivially convergent.

We use the standard Event-B refinement to gradually replace event op with a convergent, deterministic program. The determinacy of a final specification is established at the syntactic level (only deterministic variable updates are used in event specifications). The preservation of convergence is a part of the refinement method.

There is a small semantic mismatch. While we speak about operations in the terms of preconditions and postconditions, Event-B events are defined in the terms of guards and postconditions. The difference is that a guard may not be weakened during refinement while a precondition may not be strengthened. The solution is to insist that an abstract

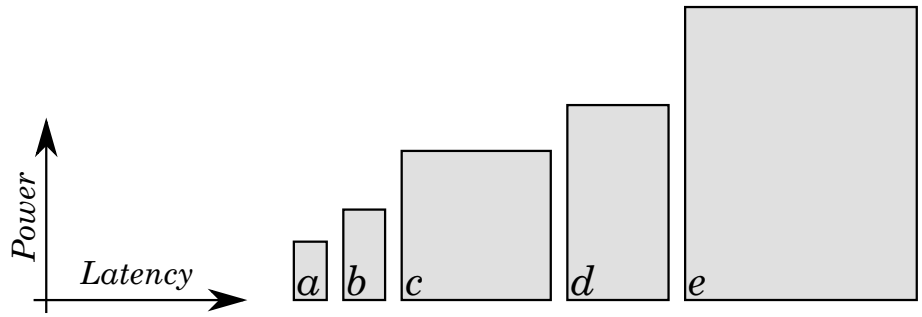


Figure 3.7: Datapath components for *DP3* implementation

event guard is always refined in such a way that abstract states characterised by the guard are all accounted for by the guards of concrete events. In other words, the collective precondition of an implementation is not more restrictive than in the abstract model:

$$I(v) \wedge G(v) \Rightarrow H_1(v) \vee \cdots \vee H_n(v),$$

where G is a guard of some abstract event and H_i are the guards of a subset of concrete events. The condition states that whenever an event is refined, for every state of the event guard there is always something to do in the refined machine.

An illustration to the described modelling approach is provided in Section 3.5.

3.5 Case study

In this section we study a common low-level GPU instruction, called *DP3*, which given two vectors $\mathbf{x} = (x_1, x_2, x_3)$ and $\mathbf{y} = (y_1, y_2, y_3)$, computes their dot product $\mathbf{x} \cdot \mathbf{y} = x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3$. There are many ways to achieve the required functionality in hardware; consider the following datapath components (denoted by $a \ldots e$) which can be used to fulfil this task:

a) 2-input adder;

b) 3-input adder;

c) 2-input multiplier;

d) fast 2-input multiplier;

e) dedicated DP3 unit.

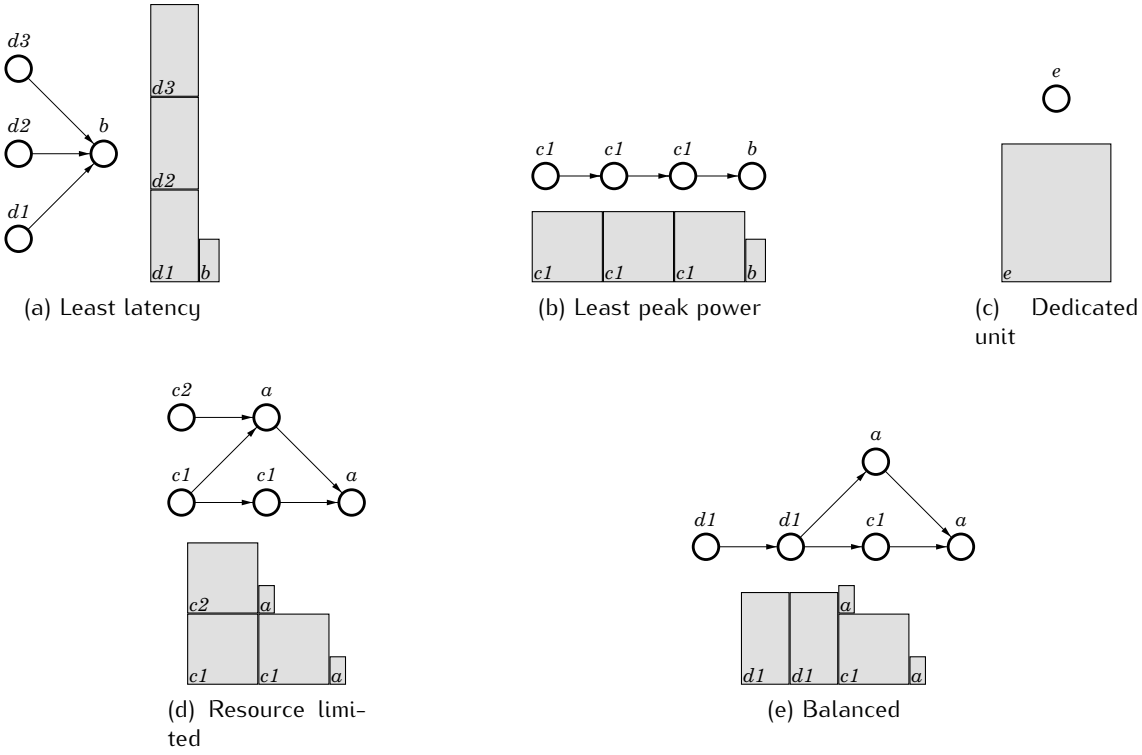
Similar to the *Energy Token model* [134], we associate two attributes, *execution latency* and *power consumption,* with every component. Figure 3.7 depicts them as labelled boxes with dimensions corresponding to their attributes; the area of a box represents the *energy* required for the computation.

Depending on the current operation mode and availability of the components, a processor has to schedule their activation in the appropriate partial order. Figure 3.8 lists several possible partial orders together with their power/latency profiles.

Least latency implementation: the fastest way to implement the instruction is to compute multiplications $tmp_k = x_k \cdot y_k$ concurrently using three fast multipliers d1–d3 and then compute the final result $tmp_1 + tmp_2 + tmp_3$ with a 3-input adder b; see Figure 3.8(a). This implementation is the costliest in terms of peak power and thus may not always be affordable.

Least peak power implementation: a directly opposite scheduling strategy is shown in Figure 3.8(b). Three multiplications are performed sequentially on the same slow multiplier c1, followed by 3-input addition b. This strategy has the largest latency among all the presented because it is completely sequential and uses slow power-saving components. On a positive side, this implementation requires only two basic functional blocks, which are likely to be reused by other instructions, so its *resource utilisation* is high.

Use of a dedicated component: it is possible that the chosen hardware platform contains a dedicated computation unit capable of computing dot product of two vectors, e.g. *Altera Cyclone III* FPGA board allows building a functional block called *ALTMULT_ADD*(3) with three multipliers connected to a 3-input adder. We can directly execute this block without any scheduling – see Figure 3.8(c). While being convenient and potentially efficient due to custom design, such a solution is not always justified because of low resource utilisation: it is impossible to reuse the built-in multipliers for implementing other instructions and if *DP3* is rarely used by software then this dedicated component will be wasting area and power (due to the leakage current) most of the time.



(a) Least latency

(b) Least peak power

(c) Dedicated unit

(d) Resource limited

(e) Balanced

Figure 3.8: Different implementations of *DP3* instruction

Moreover, such an implementation does not allow any dynamic reconfiguration thereby is less flexible.

Fast implementation with limited resources: if there are only two available multipliers c1 and c2 (either because of hardware limitations or because other multipliers are busy at the moment) then the fastest possible scheduling strategy is as follows. At first, two multiplications should be performed in parallel. Then their results are fed to 2-input adder a, while c1 is restarted for computing the third multiplication. Finally, the obtained results are added together by the same adder a as shown in Figure 3.8(d).

Balanced solution: Figure 3.8(e) presents a balanced strategy, which aims to spread power consumption evenly over time, while being relatively fast. This schedule may be advantageous for the best *energy utilisation* and in security applications.

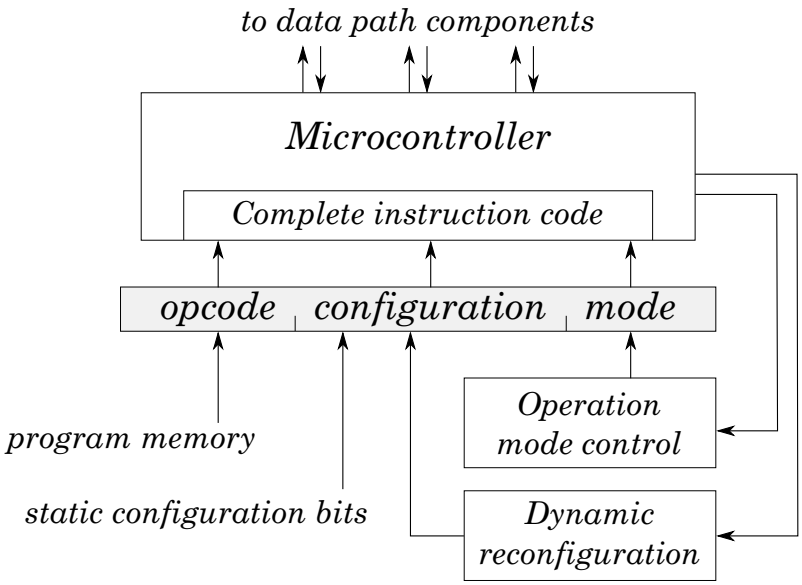
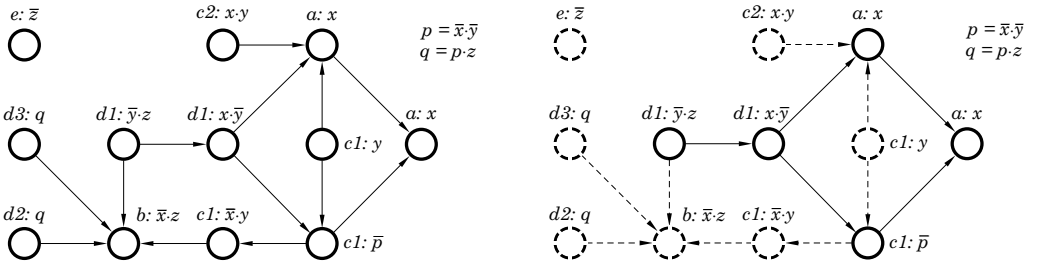


Figure 3.9: Complete instruction code

3.5.1 Derivation of the instruction set

We could devise more implementations of this instruction, but this is not the point of the case study. The goal is to demonstrate that even such a basic instruction as *DP3* has a lot of valid scheduling strategies with distinct characteristics. Importantly, it is not possible to select the best strategy because a priori it is not known which one is better. Therefore including only one of them into a processor instruction set is a serious compromise which should not be done at this early and abstract stage of the design process. We propose to include as many different implementations into the instruction set as possible, and, if needed, reduce the behavioural spectrum at the later design stages when more information is at hand (some final decisions can even be made during runtime by dynamic processor reconfiguration). The CPOG model is well suited for this task: it can represent a multitude of different implementations of the same instruction in a compact overlaid form. If the instruction is intended to have only one opcode, we can distinguish between its different implementations using *mode* and *configuration variables*. They are not part of the opcode (which is fetched from the program memory during software execution), but can be dynamically changed by the power/latency runtime control mechanisms [161] or be statically set to constants according to the limitations of the actual hardware platform, as shown in Figure 3.9.



(a) Composition H of all implementations

(b) Projection $H|_{\psi_5}$, $\psi_5 = (1,0,1)$

Figure 3.10: CPOG specification of *DP3* instruction

We can specify all the discussed implementations of *DP3* instruction using a single CPOG. To do that we first have to encode all of them. If there are no requirements on the mode/configuration codes, then a designer is free to assign them arbitrarily, however it may affect CPOG complexity and, as a consequence, complexity of the resultant microcontroller. In this case it is possible to resort to the help of automated[8] optimal encoding methods [104], which generate codes $\psi_1 = 001$, $\psi_2 = 011$, $\psi_3 = 000$, $\psi_4 = 111$, and $\psi_5 = 101$ for the five partial orders depicted in Figure 3.8 (note that these optimal codes are far from trivial sequence of binary codes 000-100). If we compose all of them into a single CPOG using the method from Section 3.3.1, we obtain the graph shown in Figure 3.10(a). The mode/configuration variables are denoted as $X = \{x, y, z\}$, and two intermediate variables $\{p, q\}$ are derived from them to simplify other graph conditions; as a result only seven 2-input gates are required to compute all graph conditions. The obtained graph is a superposition of the given partial orders, i.e. all of them can be visually identified in it – see, for example, Figure 3.10(b), which shows the balanced implementation generated by code ψ_5, and compare it with partial order in Figure 3.8(e). For a designer this gives a useful higher-level picture which brings out interactions between the components much better than separate partial order diagrams (this is similar to a metro map which represents a set of metro lines in a compact understandable form).

3.5.2 Verification of correctness

We now demonstrate the application of the Event-B modelling and verification approach described in Section 3.4 to the above example. Due to the similarity of the approach we described the least latency implementation of DP3 instruction, as shown in Figure 3.8(a). We show with a formal approach that our chosen implementation does indeed compute the dot product of two vectors. The following is a simple *DP3* instruction specification that defines only one normal case.

instruction dotp is

$c = \text{TRUE} \rightarrow r = x(1) * y(1) + x(2) * y(2) + x(3) * y(3)$

end

As the first step, we obtain an abstract Event-B state model of the instruction by instantiating the model template given above. The properties of the dot product operation are substituted in the place of abstract predicates P and Q. The result is the following Event-B machine. Note that the specification is generalised to an arbitrary vector length. This does not affect proofs and the model may be reused should there be a need for a differing vector length:

[8] We used Workcraft framework [6] for CPOG modelling and encoding.

MACHINE dotp

VARIABLES x, y, r, c

INVARIANT

$x \in 1..n \rightarrow \mathbb{Z}$

$y \in 1..n \rightarrow \mathbb{Z}$

$r \in \mathbb{Z}$

$c \in \mathbb{B}$

$c = \text{TRUE} \Rightarrow r = \Sigma\{x(i) * y(i) \mid i \in 1..n\}$

INITIALISATION

$x :\in 1..n \rightarrow \mathbb{Z}$

$y :\in 1..n \rightarrow \mathbb{Z}$

$r :\in \mathbb{Z}$

$c := \text{FALSE}$

EVENTS

dotp = when

$c = \text{FALSE}$

then

$r := \Sigma\{x(i) * y(i) \mid i \in 1..n\}$

$c := \text{TRUE}$

end

END

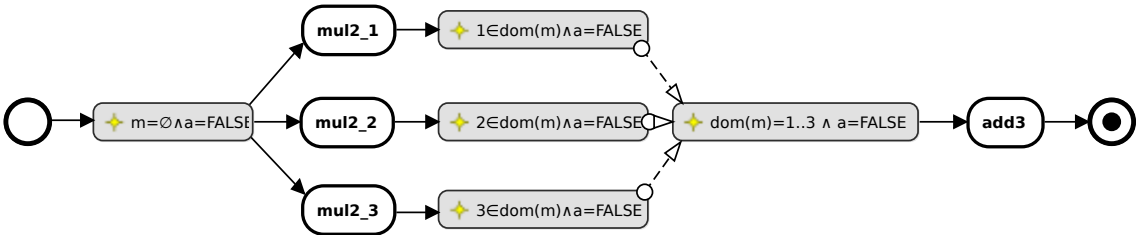
The machine is refined into an implementation that makes use of n parallel multipliers and one n-input adder; this is a generalised version of the least latency implementation. The result is the model shown in Figure 3.11.

All the consistency and refinement proof obligations are discharged by autonomous theorem provers. Once a concrete model of an instruction is developed and verified it must be, somehow, transformed into a graph to feed it into the CPOG synthesis routines. For this we construct a graph expressing possible event orderings (called the flow aspect of a model). This additional model must be proven consistent with the Event-B machine in a sense that all the paths in such a graph are also possible event sequences in the history of a machine execution. The relevant proof obligations are generated automatically by the Event-B modelling tool [148]. The following flow aspect is constructed for a trivial specialisation of least_latency where $n = 3$ with parametrised event mul3 split into three

```
MACHINE least_latency
  refines dotp
  VARIABLES x, y, r, c, m
  INVARIANT
      m ∈ 1..n ⇸ ℤ
      ∀i · i ∈ dom(m) ⇒ m(i) = x(i) ∗ y(i)
  INITIALISATION ... ‖ m := ∅
  VARIANT 1..n \ dom(m)
  EVENTS
      mul2            =   any i where
                              i ∈ 1..n
                              i ∉ dom(m)
                          then
                              m(i) := x(i) ∗ y(i)
                          end
      addn ref dotp   =   when
                              c = FALSE
                              dom(m) = 1..n
                          then
                              r := Σ(m)
                              c := TRUE
                          end
END
```

Figure 3.11: Machine for the least latency implementation

separate events, one for each $i \in \{1,2,3\}$; the n-input adder becomes 3-input adder:



The shaded boxes are assertions — elements aiding in the construction of a proof; these do not contribute to the output control graph. Single and double circles are the initialisation and termination actions; the rounded boxes are the events of the machine. The input for CPOG synthesis is a graph obtained by removing assertion elements and dropping all the edge and node annotations. Other implementations of the *DP3* instruction can be verified in a similar way.

Event-B Rodin Platform employs a range state-of-the-art verification techniques based on automated theorem proving, constraint solving and model checking. This makes it possible to discharge around 80% to 95% of all verifications conditions completely automatically. For more involved cases and to study failed proofs, Rodin also features an interactive proof environment and a library of rewrite (simplification) rules. This allow a

user to provide proof hints or build a proof skeleton with details filled in by automated tools. From large-scale projects, it was estimated that a verification engineer works at an average pace of 12-20 proof obligations per day doing one or two interactive proofs. A medium size model is about 600 - 1400 proof obligations, takes around three months and the actual proof effort is small proportion of it.

3.6 Conclusions

In this chapter we discussed main stages of design of instruction set architectures for a microprocessor. It was shown that one of the key difficulties is the necessity to comprehend and deal with a large number of instructions, whose microcontrol implementation may be altered to suit a particular hardware platform or a particular operating mode.

We demonstrated that the Conditional Partial Order Graph model is a convenient and powerful formalism for specification of processor instruction sets. It is possible to efficiently describe many different 'microcode' implementations of the same instruction as a single mathematical structure and perform its refinement, optimisation, and encoding using formal CPOG transformations. Crucially, these transformations operate on a CPOG specification rather than on the instruction set itself and thus their complexity does not depend on the number of different instructions.

The overall number of CPU instructions is often quite large although the majority of them are of a fairly trivial nature. To free a designer from the tedium of attending to the minute details of instruction logic we plan to implement a procedure to automatically construct a collection of correct instruction specifications. A number of such procedures were studied within the constructive logic where the proof of a specification statement is given in terms that permit an automatic extraction of an executable program. Although the search space for a proof is potentially large, the application of proof planning techniques, such as rippling and abstraction, reduce it considerably to make possible the discovery of non-trivial programs with loops and branching [28].

In the last section of this chapter we demonstrated an example of a simple DP3 instruction, which shows how CPOGs can be used for capturing different hardware configura-

tions and operation modes and to formally reason about correctness of CPOG constructs with respect to the given functional ISA descriptions using the Event-B model.

The presented partial order representation and functional verification of instruction sets were applied to a more sophisticated instruction set in the next chapter, where we used it for the design of a microcontroller of a new asynchronous Intel 8051 microprocessor.

In this chapter we addressed compositional design flow, which is currently fully automatic. Designer only needs to feed a PO representation to the tool (the Workcraft tool with SCENCO plugin), which then automatically encode them and overlay into the final CPOG.

Chapter 4

Design of an Asynchronous 8051 Microprocessor

In the previous chapter we described the main aspects of designing microprocessor instruction sets and presented a case study to show the benefits of the introduced compositional approach. To demonstrate our methodology on a more sophisticated example and to introduce a new power-proportional criterion in the system design flow, as we discussed in Chapter 1, we implemented a new asynchronous 8051 microprocessor. Its complete design flow and specific implementation details are described in this chapter.

During the development process we formulated a new flow for the processor implementation [132], which is shown in Figure 4.1. This flow can be divided into two main stages: Design and Implementation.

The *Design stage* is usually the initial point from where the development of a CPU or any other complicated system starts. This part is mainly focused on the primary understanding of the main aspects of microprocessor development:

- application specific aspects, such as the microprocessor's architecture, instruction set, application specific functionality, etc.
- non-functional aspects, such as environmental conditions (e.g. support for unstable voltage supply, wide range of operating temperatures, etc.), support for a hierarchy of energy-saving modes (high performance, low power), etc.

This stage can be divided into several steps: analysis of the processor architecture and instruction set (Section 4.1), and specification of the control logic and datapath components (Section 4.2).

The *Implementation stage* includes the development of the central microcontroller, datapath components and interface protocols.

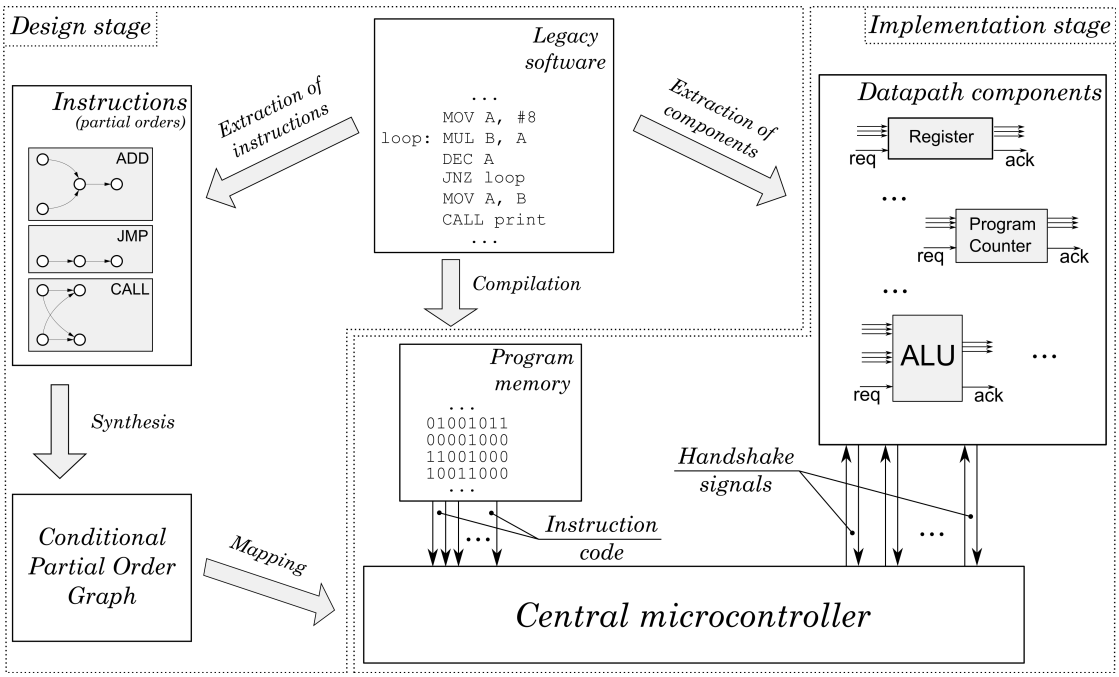


Figure 4.1: Conceptual view of the design process

In Section 4.2 we discuss the implementation of two main control logic blocks for the microprocessor: the top level (Section 4.2.1) and the ALU (Section 4.2.2) control logic block. We make use of the previously presented compositional approach (Section 2.2), which is highly beneficial for systems with many behavioural scenarios defined on the same set of events and actions, such as the control logic of a CPU [9].

The structure of the CPU datapath (internal implementation details and specific features) and the communication protocol between control and datapath units are described in Section 4.3.

Finally, Section 4.4 discusses the applied ISA transformations and optimisations (such as extended datapath structure, adjustable delay lines, issues of fault tolerance, etc.) and Section 4.5 introduces design for test features, which were applied during the implemen-

tation.

4.1 Asynchronous 8051 architecture and instruction set

As was mentioned in Chapter 2.3, the Intel 8051 architecture is still popular and used in various devices, embedded systems and in a wide range of applications. Our implementation follows the Harvard architecture [45] of the original Intel 8051 CISC (Complex Instruction Set Computer) and supports 257 instructions of the microprocessor (255 standard instructions and two extra instructions specific to our implementation (Section 4.4.3 and Appendix A)). The top-level architecture of the CPU as well as the full instruction set are described in the Philips 80C51 Data Handbook [48], which contains about 100 pages of the architecture specification and about 1300 pages of various derivatives of the 80C51. In this section we describe several significant changes we introduced in our implementation:

- We chose the asynchronous design style for our implementation. There are numerous advantages of the self-timed approach, as discussed Section 2.1. Figure 4.2 and Figure 4.9 show that the control circuitry communicates with datapath units by means of request and acknowledgement signals, which use a 4-phase handshake protocol (Section 4.3.8).

- We implemented a more ambitious 16-bit version of the Harvard[1] architecture to obtain higher performance, by using a unified 16-bit width for both address and data buses.

- The 16-bit datapath was extended with additional computational units (adder, multiplier and divider), which were specifically optimised to work in a particular operational mode (Section 4.4.1) . By using this approach we achieved robust operation of the circuit over a wide range of supply voltages.

- We can easily choose which computational unit to use depending not only on application or environmental aspects, but also on the functional correctness of an

[1] Instruction memory and data memory are separate.

individual unit, thus addressing the issue of fault tolerance (Section 4.4.3). This was done by adding a special internal register "Unit Selector" (see Figure 4.9) to the architecture. By accessing this register, we can specify which computation units are operational and which are not.

The specification of a complex system such as a microprocessor usually starts at the architectural level [133][100], where the structural abstraction enables a designer to divide the system into several subsystems, thus significantly simplifying the design flow and reducing the solution search space. It is particularly important to refine the architecture to the level of operational units (or datapath) and behavioural scenarios (or control logic). There are several ways in which this can be done:

- Refinement from software (written in C/C++, Assembly language, etc.).
- Using Architecture Description Languages (ADLs), such as structural ADLs (e.g. MIMOLA [166], UDL/I [75]), behavioural ADLs (e.g. nML [57], ISDL [66]), mixed ADLs (e.g. LISA [74], EXPRESSION [67]) and partial[2] ADLs (e.g. AIDL [65]).
- Other instruction specifications, such as a list of instructions obtained from a microprocessor specification (e.g. a CPU manual). This type of specification is not formal and therefore should be processed manually.

As we had a full description of the microprocessor, its instruction set and architecture details, we proceeded with a manual specification.

Figure 4.2 shows the block diagram of the top-level architecture of the microprocessor. The top level hierarchy of any CPU (including ours) can be divided into two parts: *Control*, which contains Control logic and *Datapath* – the rest of the blocks. Note that the main datapath block – Arithmetic Logic Unit (ALU) (Figure 4.9) is a complicated system with a hierarchical structure. Therefore we treated it in the same way as the whole system, and refined it further down to the level of operational units and behavioural scenarios.

[2]Usually ADLs are divided into three categories, depending on the nature they capture the structure: structural – the architectural components and their connectivity; behavioural – instruction set behaviour; mixed – both structure and behaviour of the architecture [101].

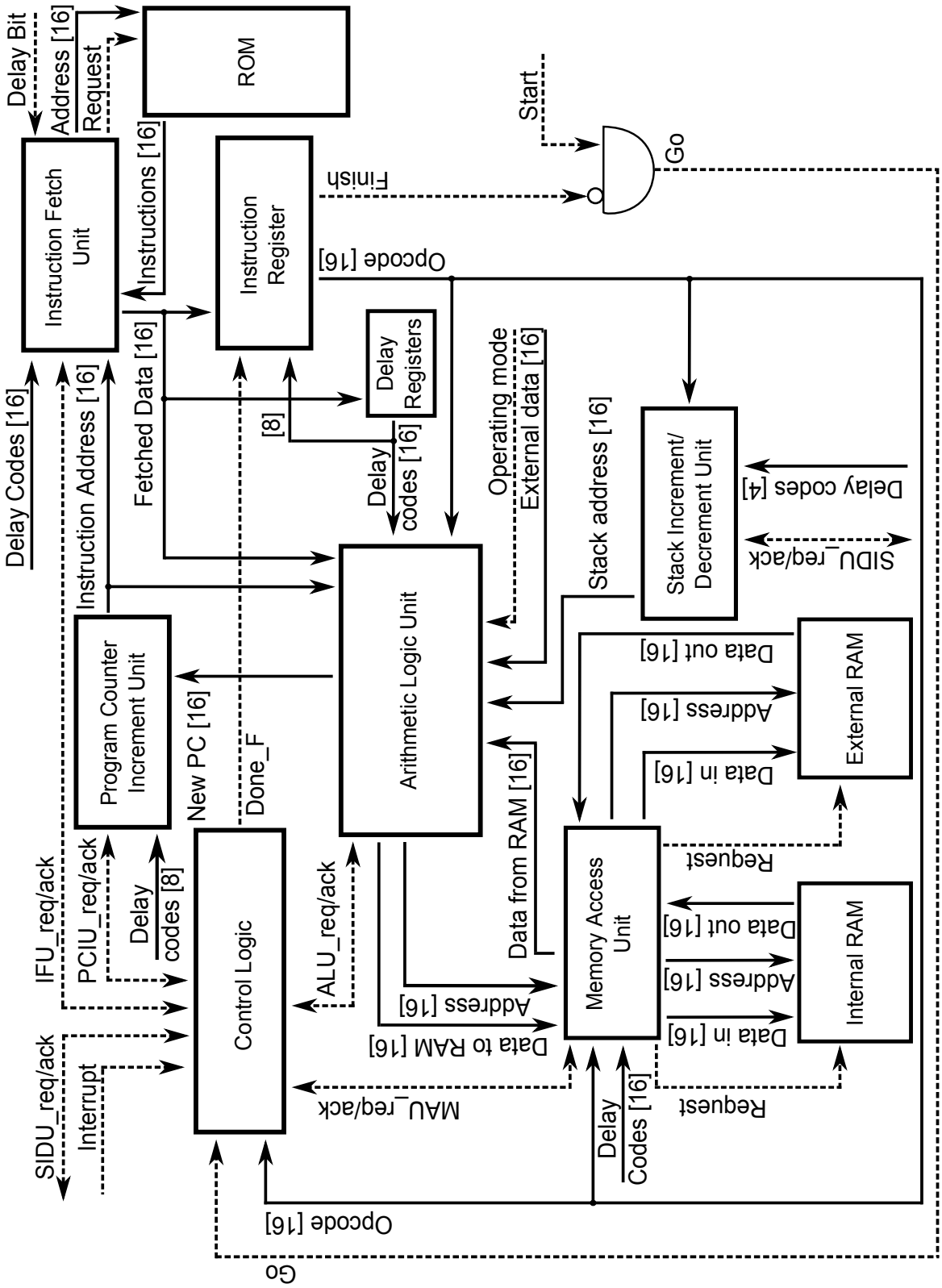


Figure 4.2: Architecture of the proposed microprocessor (dashed lines represent a 1 bit wide signal line, the width of other connections is shown in brackets)

Sections 4.2 and 4.3 describe the structure, functionality and implementation flow of each block from the diagram in detail.

4.2 Specification of the control logic

The control logic of a microprocessor coordinates its components and directs the execution of instructions. Our implementation of the Intel 8051 processor contains 257 instructions; to cope with such a complex ISA it was essential to make use of the presented compositional approach. Two control logic blocks were developed in our implementation: the top level control logic and the ALU control logic. In the following subsections we discuss their implementation and structure in detail.

4.2.1 Top level control logic

The top level control logic is the main control logic of the microprocessor. It coordinates the execution of all CPU datapath components from the top architectural level.

The key part in a microprocessor control specification is the description of instructions. Each instruction corresponds to a schedule of primitive actions such as data transfer, arithmetic operation, memory access, etc., which are performed by datapath components. The control logic design flow contains the following steps:

Extraction of datapath components. In the previous section we have already stated several ways in which operational units (or datapath) and behavioural scenarios can be extracted. Table 4.1 presents five functional components extracted from the 8051 instruction set [80] specification. This type of specification is not formal, therefore was processed manually. More detailed information about datapath components is given in Section 4.3. We proceed to the specification of the individual instructions.

Extraction of partial orders for instructions. PO is essentially a model of ordering of actions, with associated cause and effect relationships [102].

Each instruction was specified as a PO (see Appendix A). Some of the instructions had the same PO, so we eventually grouped them into classes. Finally we obtained 37 different classes of different dependency graphs, with the largest class containing up to

Table 4.1: Function components extracted from ISA

Components	Description	Duplicate usage of the same component
PCIU	Program Counter Increment Unit is responsible for incrementing the program counter (instruction pointer) to indicate where the execution is in a program source code.	PCIU/2 and PCIU/3
IFU	Instruction Fetch Unit extracts operational code of the instruction from the program memory using the provided address pointer.	IFU/2, IFU/3 and IFU/4
MAU	Given an address pointer and input data, the Memory Access Unit accesses the data from internal and external RAM.	MAU/2, MAU/3, MAU/4, MAU/5 and MAU/6
ALU	Arithmetic Logic Unit is the main computation unit, which performs arithmetic and logical operations in the CPU. It also provides appropriate addresses and data for MAU and IFU in specific instructions.	ALU/2, ALU/3, ALU/4, ALU/5, ALU/6 and ALU/7
SIDU	Stack pointer Increment/Decrement Unit, as the name suggests, is in charge of incrementing and decrementing the stack pointer.	SIDU/2

60 instructions.

As an example Figure 4.3 shows the PO representation of the instructions from class E^3, which corresponds to a group of 17 instructions having the same PO representation:

- ***MOV direct, A*** In this instruction, access to internal RAM (execution of components $ALU \rightarrow MAU$) is performed concurrently with fetching a target address (*direct*) into the Instruction Register (IR) $PCIU \rightarrow IFU$. Then $ALU/2$ (preparation for writing the data from the accumulator (A) into the target RAM location) is executed concurrently with an increment of Program Counter ($PCIU/2$). Finally, it is possible to write data from the accumulator into the target memory location ($MAU/2$) and fetch the next instruction into IR ($IFU/2$).

- ***MOV direct, Rn*** This group of instructions is similar to the previous one with the difference that we are moving data from an internal register (Rn) to a target internal RAM location (*direct*). There are 8 different instructions, as we have access to 8 different registers (R0-R7) in the register bank.

- ***MOV @Ri, #immediate*** In this group of instructions one of the operands is an

[3]All the classes are specified in an alphabetic order, see Appendix A.

internal RAM memory location addressed indirectly and the other is an immediate constant. First, access to an internal register (*Ri*) is performed concurrently with fetching a direct constant (*#immediate*) into the IR. Then ALU/2 (preparation of writing the data in IR to address given in *Ri*) is executed concurrently with an increment of PC. Finally, we write data into memory and fetch the next instruction into IR. Internal RAM can only be addressed indirectly through registers R0 or R1, so there are 2 different instructions in this group.

- ***ADD A, #immediate*** Here we have a group of ALU operations. This particular instruction performs addition of the accumulator and an immediate constant. Similarly to previous examples, we read data from the accumulator while concurrently fetching an immediate constant. Then we execute the operation of addition (ALU/2) concurrently with an increment of PC, and finally write data into memory and fetch the next instruction into IR. It can be noticed that other instructions have the same behaviour: *ADDC A, #immediate*, *SUBB A, #immediate*, *ORL A, #immediate*, *ANL A, #immediate* and *XRL A, #immediate*, hence this group contains 6 instructions.

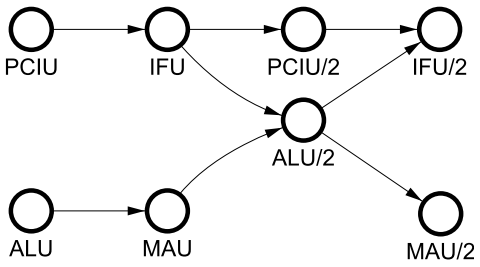


Figure 4.3: PO representation of the instructions from class *E*

Now let's look into a more sophisticated example. The previous example has no conditional behaviour. However, such instructions are also popular in programming and show all of the benefits of the compositional approach. For example, instructions from class *AE* (see Figure 4.4) contain two conditional branch instructions:

1. ***CJNE @Rn, #immediate, offset*** The CJNE instruction compares contents of the memory location whose address is provided in the specified register with a given immediate constant, and branches to the specified destination (by adding the

given address offset to the PC) if their values are not equal. Otherwise, execution continues with the next instruction. CJNE is a good example to demonstrate the compositionality of CPOGs: the complete behaviour of the instruction is split into two scenarios, which are easier to specify separately; the scenarios are then composed, resulting in the complete instruction specification.

- Figure 4.4(a) shows a graph describing the order of activation of the functional units in the first CJNE scenario, where the branch is not taken because the compared values are equal. This scenario begins with two concurrent sequences of actions: $PCIU \rightarrow IFU$ is executed to fetch the constant stored immediately after the instruction opcode, while actions $ALU \rightarrow MAU$ are performed to fetch the contents of Rn from the internal memory. After that, another similar sequence is performed, $ALU/2 \rightarrow MAU/2$, to look up the contents of the memory at the address loaded from Rn. Finally, $ALU/3$ is performed to compare the obtained values; the corresponding status flags are set according to the result. In particular, if the values are equal the flag z is set to 1. In this scenario we assume that the values are indeed equal, therefore, the processor may proceed with the next instruction, that is, the program counter is incremented twice (skipping the branch offset) and the next instruction opcode is fetched (actions $PCIU/2 \rightarrow PCIU/3 \rightarrow IFU/3$).
- The second scenario, see Figure 4.4(b), is identical to the first one until the moment when comparison is performed by $ALU/3$ and it is determined that the compared values are different. At this point, the execution continues as follows. The branch offset is loaded by performing $IFU/2$ straight after $PCIU/2$. Then the actual branch operation is executed by adding the offset to the current PC value ($ALU/4$) and fetching the next instruction opcode. Note that action $PCIU/3$ is skipped in this scenario.

2. ***CJNE A, direct, offset*** The only difference between this instruction and the previous one is that this one compares contents of two memory location, one of which

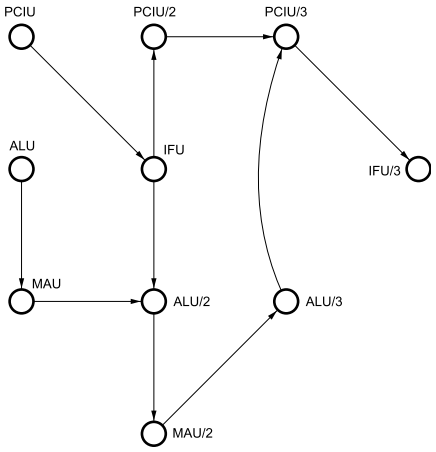
is located in the SFR – Accumulator and the other one is a different internal RAM location. Actions $ALU \rightarrow MAU$ are performed to fetch the contents of the Accumulator and $ALU/2 \rightarrow MAU/2$ to read the other internal RAM location. The rest of execution is identical to the first CJNE instruction.

In terms of the top level control execution both of the instructions are the same. However, one should notice that the data which is fetched during the actions $ALU \rightarrow MAU$ in each of the instructions needs to be treated differently: in the first one it is a memory address, which needs to be looked up; in the second one it is a value which needs to be compared. This difference is handled at the ALU control level and discussed in details in Section 4.2.2.

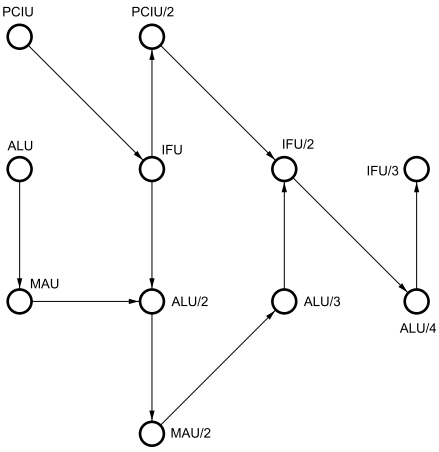
All the other instructions were represented as POs in the same manner, see Appendix A.

After we extracted POs for all the instructions it was important to show that each of the behavioural scenarios conformed to the instruction specification, i.e. they required a formal proof of correctness (see Section 3.4). This was done in the same way as we showed with the DP3 instruction (see Section 3.5). An important feature of the CPOG approach (Section 2.2) is that if we have an instruction with conditions (like ***CJNE*** instruction) and we verify the correctness of all conditions, it becomes possible to merge them into one instruction by using the CPOG composition (Figure 4.4(c)). The final instruction is correct by construction and therefore does not require any additional verification. One can see that the composition (Figure 4.4(c)) has only three conditional elements, namely, $\phi(PCIU/3) = z$ and $\phi(IFU/2) = \phi(ALU/4) = \bar{z}$. All the other vertices and all the arcs are unconditional due to the similarity between the two scenarios.

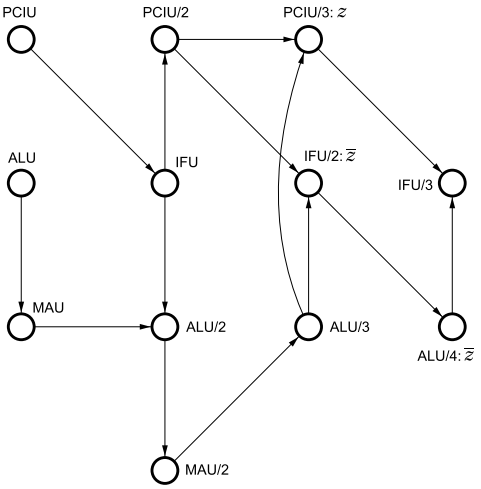
Encoding of partial orders. In order to distinguish between the synthesised POs, we need to encode them. The codes can either be assigned arbitrarily or can be provided as part of the system specification. Importantly, the size and latency of the final microcontroller circuit depends significantly on the chosen encoding of the scenarios [102]. There are several types of encoding that can be used, such as one hot, matrix, balanced [103] and others. A new technique for optimal encoding was recently introduced [104], but



(a) Specification of the first scenario of CJNE ($z = 1$)



(b) Specification of the second scenario of CJNE ($z = 0$)



(c) Complete specification of CJNE instruction

Figure 4.4: CPOG specifications of CJNE instruction

due to the complexity of the optimal encoding algorithm it was unable to process 257 instructions, which we have in our implementation.

Therefore we had to use a semi-automated approach based on the Huffman encoding algorithm [76] because of its simplicity and speed. It reduces the number of bits required to encode instructions. The complete Huffman encoding tree is shown in Figure 4.5.

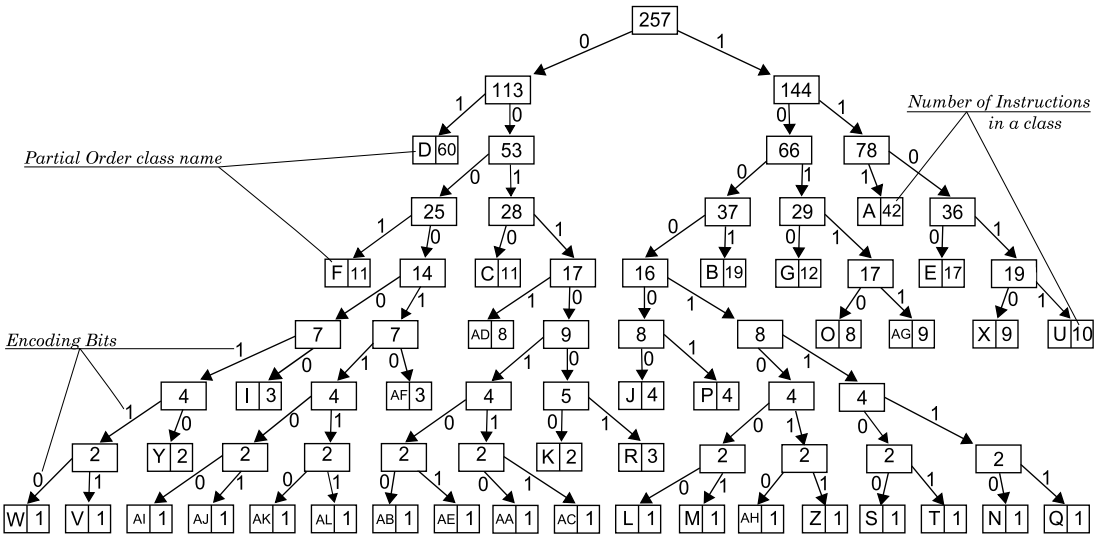


Figure 4.5: Representation of Huffman Encoding tree of Partial order classes

It is one of our current work to analyze the effect of the encoding on the area and performance of the final controller.

CPOG generation. Now, after we represented all the instructions of the CPU as POs and encoded them, it became possible to synthesise a CPOG containing all of them, as shown in Figure 4.6. Note that most of vertexes and arcs have conditions, depending on which particular PO can be activated or disabled in accordance with the evaluation of the conditions under the *opcode*. The obtained graph is a superposition of the given partial orders, i.e. all of them can be visually identified – see, for example, Figure 4.7(a), which shows the projection of PO for the instructions from class D and Figure 4.7(b) – from class Y. In the same way, each of the POs can be activated in this CPOG. For a designer this gives a useful higher-level picture which brings out the interactions between the components much better than separate partial order diagrams (this is similar to a metro map which represents a set of metro lines in a compact understandable form). At his stage it is not only possible to verify the correctness of PO encoding (the correspondence

between opcodes and a needed POs), but also to check if a particular PO is correct (correctness of vertices and arcs conditions). Both of these verification checks can be done using the Workcraft framework [6].

Mapping. The final stage of our control logic design is the mapping of the CPOG representation into a set of logic gates. As soon as the CPOG specification of a system is synthesised and checked for correctness, it can be mapped onto Boolean equations in order to produce a physical implementation (gate-level netlist) of the specified micro-controller. The mapping procedure is a purely structural operation and was addressed in Section 2.2 and Section 3.3.4.

Finally, we can translate the obtained Boolean equations into VHDL, Verilog or other HDL and/or input these equations into technology mapping and Place and Route (P&R) tools (e.g. Synopsys Design Vision [142], Cadence Encounter Digital Implementation System [30], etc.). See the demonstration chip example in Section 5.2 and Appendix B.1 section for all the resulting Boolean equations.

Along with hardware mapping we have to perform software mapping, i.e. the compilation of the program code from a given legacy software and store the compilation result in the program memory.

Our design process flow, defined in Figure 4.1, shows that the interface between control logic and datapath components is based on a handshake (request-acknowledgement) protocol. This allows significant flexibility in reusing the datapath components, such as ALU, PCIU, IFU, memory block, etc., by the controller. So each of these components can be executed several times with different functionality during the execution of a particular instruction depending on the order when a particular component was requested. Due to the handshake protocol, the full power of partial orders can be exploited, because the timing of control events is not bounded to particular delay constraints. The advantages of such an approach have been recently applied to the designs in [19][55][38]. All details of the number and types of the CPU components, such as registers, program counter, ALU, etc., are described in Section 4.3.

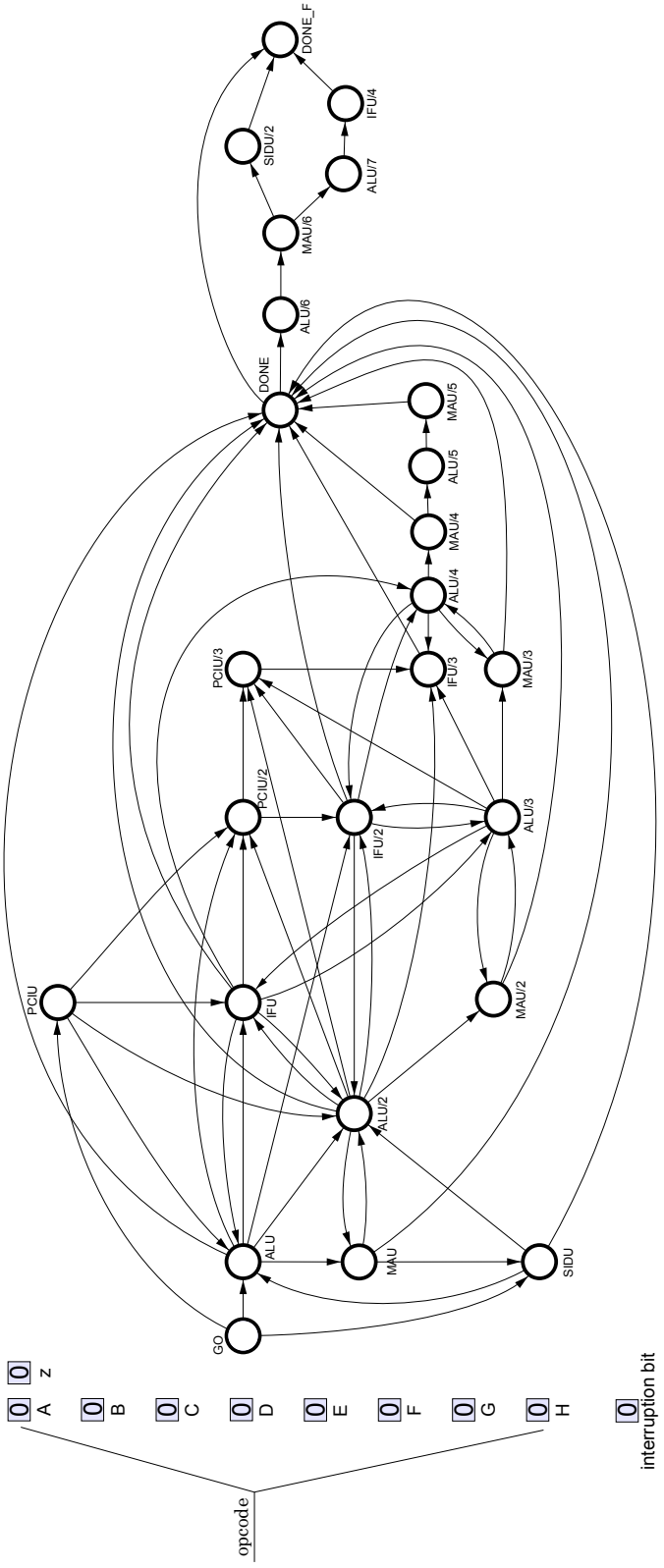
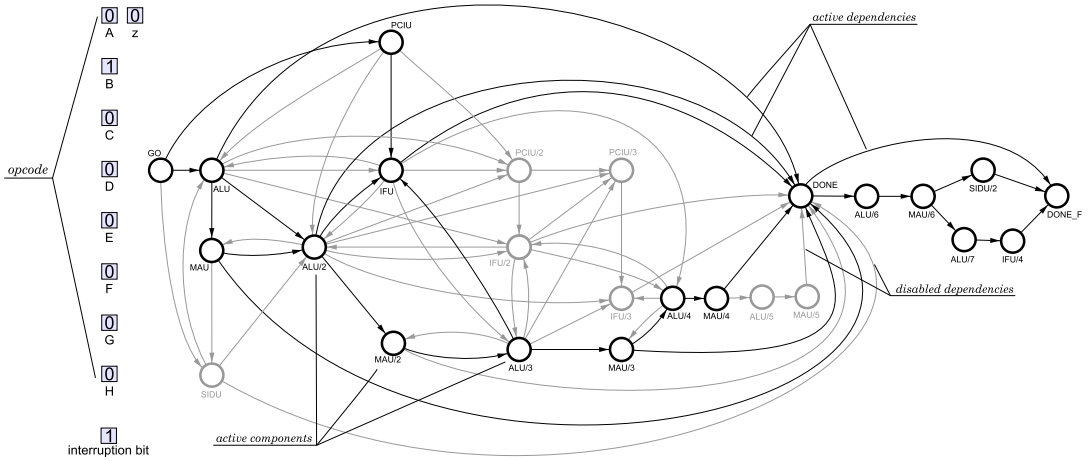
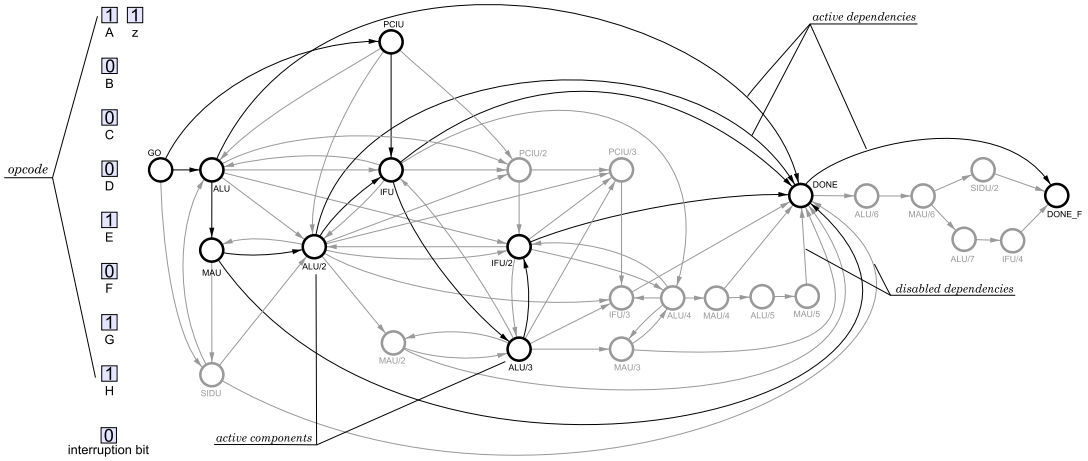


Figure 4.6: Complete instruction set in CPOG representation



(a) Projection D with interrupt



(b) Projection Y without interrupt

Figure 4.7: Examples PO Projections in the whole instruction set

4.2.2 ALU control logic

Unlike the top level control logic, the ALU control logic coordinates the execution of the datapath only within the Arithmetic Logic Unit. As was mentioned in Section 4.1, the ALU itself is a complicated circuit with a hierarchical structure, so a dedicated circuit was developed to control the computational part of the ALU (Figure 4.9).

Control logic of the ALU block was designed using the same flow (Figure 4.1) which was applied to the development of the top-level control logic. However, the flow was focused not on the top-level architecture of the CPU and instruction set, but on the specific ALU features, such as the availability of specific datapath components, the operating

Table 4.2: Specification of functioning components in ALU

Components	Description
Address_R and Data_R	Address Register and Data Register are used to keep address and data for accessing internal and external memories.
Temp_R and Temp2_R	Two Temporary Registers are reserved for internal usage.
Datapath	The main computational unit in this block. It is used for all arithmetic and logical operations in the CPU.
PSW_R	PSW (Program status word) Register is used to keep information about processor's status and required for proper program execution.
PC_R	Program Counter Register holds address of the next instruction.
Unit Selector_R	Unit Selector Register keeps information about operating arithmetic units.

modes, the order of ALU component execution, etc.

Following the flow we extracted the datapath components (see Table 4.2) needed for the correct execution of the ALU's behavioural scenarios. Also, additional hardware elements were added, such as the "Work" Register for fault tolerance control, which can also be found in Table 4.2.

The next step was the extraction of behavioural scenarios of the ALU unit in a particular instruction. Essentially we used the same instruction groups, as for the top level control (see Appendix A). However, depending on the order in which the ALU component was requested during the instruction execution the control logic behaves differently, therefore in the PO representation we introduced parameters ALU-ALU7 in addition to the opcode.

At the encoding stage we simply extended the existing opcode from the top level control logic of instructions groups (see an example of opcode for group A in Table 4.3). Note that each instruction group has its own breakdown of the opcode, due to the different number of instructions per group and the therefore varying length of the opcode (see Appendix A).

After all POs were extracted and encoded we eventually combined them into a CPOG (Figure 4.8), which is not as complex as the top-level control logic (Figure 4.6), due to

Table 4.3: Breakdown of opcodes for instructions from group A

Instruction name	Main opcode[1]	(bits 12..11)[2]	Rn number[3]	(bits 7..6)[2]	Reading from[4]	Writing to[4]	Instruction opcode
MOV A, Rn	111	11	000	00	010	000	1111100100010000
MOV Rn, A	111	11	000	00	000	010	1111100100000010
INC Rn	111	00	000	00	010	010	1110000100010010
INC A	111	00	xxx	00	000	000	1110000000000000
INC DPTR	111	00	xxx	00	100	100	1110000000100100
DEC Rn	111	00	000	01	010	010	1110000101010010
DEC A	111	00	xxx	01	000	000	1110000001000000
RLC A	111	00	xxx	10	000	000	1110000010000000
RRC A	111	00	xxx	11	000	000	1110000011000000
RL A	111	01	xxx	00	000	000	1110100000000000
RR A	111	01	xxx	01	000	000	1110100001000000
DA A	111	01	xxx	10	000	000	1110100010000000
SWAP A	111	10	xxx	00	000	000	1111000000000000
CPL A	111	10	xxx	01	000	000	1111000001000000

1. The main opcode (bits 15..13) is generated using the Huffman encoding tree for the top level control logic.
2. Bits 12..11 and 7..6 are used to distinguish instructions within the group.
3. Rn number (bits 10..8) identifies the specific register (R0-R7) from the Register Bank [80]. "xxx" description means that the Register Bank is not used for executing that instruction.
4. "Reading from" and "Writing to" are two areas in the opcode, which show the source from where the data is read and destination where it should be written: "000" – accumulator, "001" – register *B*, "010" – register from the Register Bank, "100" – DPTR, "011" – PSW [80].

the fact that there are mainly "request to the datapath unit" operations, and many of these primitive actions are happening concurrently. Again we don't show vertex and arc conditions on the diagram for clarity.

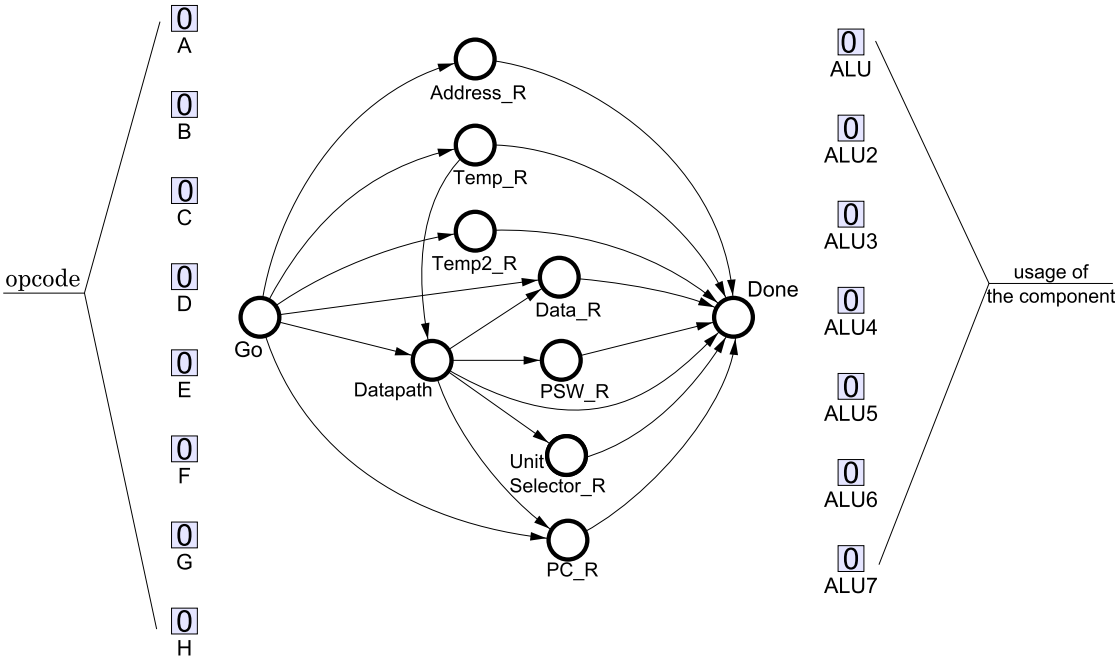


Figure 4.8: CPOG representation of ALU control

We used the Workcraft framework [6] to simulate and verify the correctness of ALU CPOG functionality.

One can notice that we addressed the verification process in Section 3.4.1 using Event-B formalism. Instruction verification using Workcraft tool and Event-B formalism are different approaches targeting different things. The Workcraft framework was used to create each of the PO, encode and produce the final CPOG representation. Further on we use this tool for a functional simulations of POs. However Event-B formalism is use for a formal verification of created PO, i.e it checks how an instruction is assembled from smaller blocks and whether such an assembly always delivers right results in all the possible instruction executions.

Finally we translated the obtained CPOG into Boolean equations (see Appendix B.2) using techniques described in Section 4.2.1. Then these equations were passed to a synthesis tool (e.g. Synopsys Design Vision [142], Altera's Quartus II FPGA design soft-

ware [53], etc.) for obtaining the gate level implementation (see Section 5.2).

Using the same design flow (Figure 4.1) for both control logic blocks (the top level and the ALU control) significantly simplified and accelerated the development process. By using such a compositional approach ISA can be easily adjusted to a needed application and/or environment conditions (see Section 3.5).

4.2.3 Interpretation using Parameterised Graph

In this Section we showed how powerful and convenient the CPOG methodology can be in control logic synthesis. Further research shows that this approach can be extended in several ways:

- Extension of a graph model representation from partial orders to general graphs. However, this extension should not exclude the usage of POs if it provides better results for needed behaviours of a system.
- Description of the equivalence relation between specification as a set of axioms and further generalisation in an algebra. This set of axioms can be proven to be minimal, sound and complete.
- By using this new algebra we provide the ability to manipulate the specifications as algebraic expressions. In other words, we are adding a syntactic level to the semantic representation of specifications similar to Boolean algebra and digital circuits.

These extensions were introduced as Algebra of Parameterised Graph (PG) [105]. This approach was implemented using a Domain Specific Language (Haskell [71]) to synthesise the control logic. Despite the fact that this is ongoing work, some examples of algebraic equations are presented in Appendix C.

4.3 Datapath description

This section outlines details of the functionality, development process and main features of the second part of CPU – the Datapath. The datapath of a processor contains arithmetical

circuits (where the actual computation takes place) registers, and dedicated memory blocks (where the data is stored), and communication paths (which provide links between them).

The datapath can be found in the microprocessor's top level structure in Figure 4.2, where it is shown around the control logic block. The rest of the section discusses the datapath components in detail.

4.3.1 Arithmetic Logic Unit

The ALU is the most important component of the datapath. As a part of the CPU architecture it was first mentioned by John von Neumann [117] in 1945. Since then, this part of a CPU has become more and more complex, and now, in high performance Graphics Processing Units (GPUs), we may have 1000s of ALU cores on the same chip.

In our implementation we were following the original ALU architecture of the 8051 microprocessor, however, we applied some special features, which we will discus in Section 4.4. As was already mentioned in Section 4.1 the ALU is a complex circuit with a hierarchical structure, shown in Figure 4.9 with its own control circuitry and a datapath.

As the control logic (*Main control* block) was already addressed in Section 4.2.2, here we concentrate on the datapath, which mainly contains computational units and internal registers:

- *Address* and *Data Registers* – two 16-bit registers, which are used for reading and writing from/to the RAM memory blocks. Both of them are connected through the *Cross bar switch* to other component, e.g. data from computational units, data from ROM or RAM blocks, etc.

- *Temp* and *Temp2 Registers* are two 16-bit registers, used for internal purposes, e.g. to exchange data between two locations. They are also connected to the *Cross bar switch* and it is possible to transfer the data to the *Datapath* block.

- *Datapath* contains arithmetic and logic operational units (see Figure 4.10). The computational units are 16-bit wide and some of them (adder, multiplier and divi-

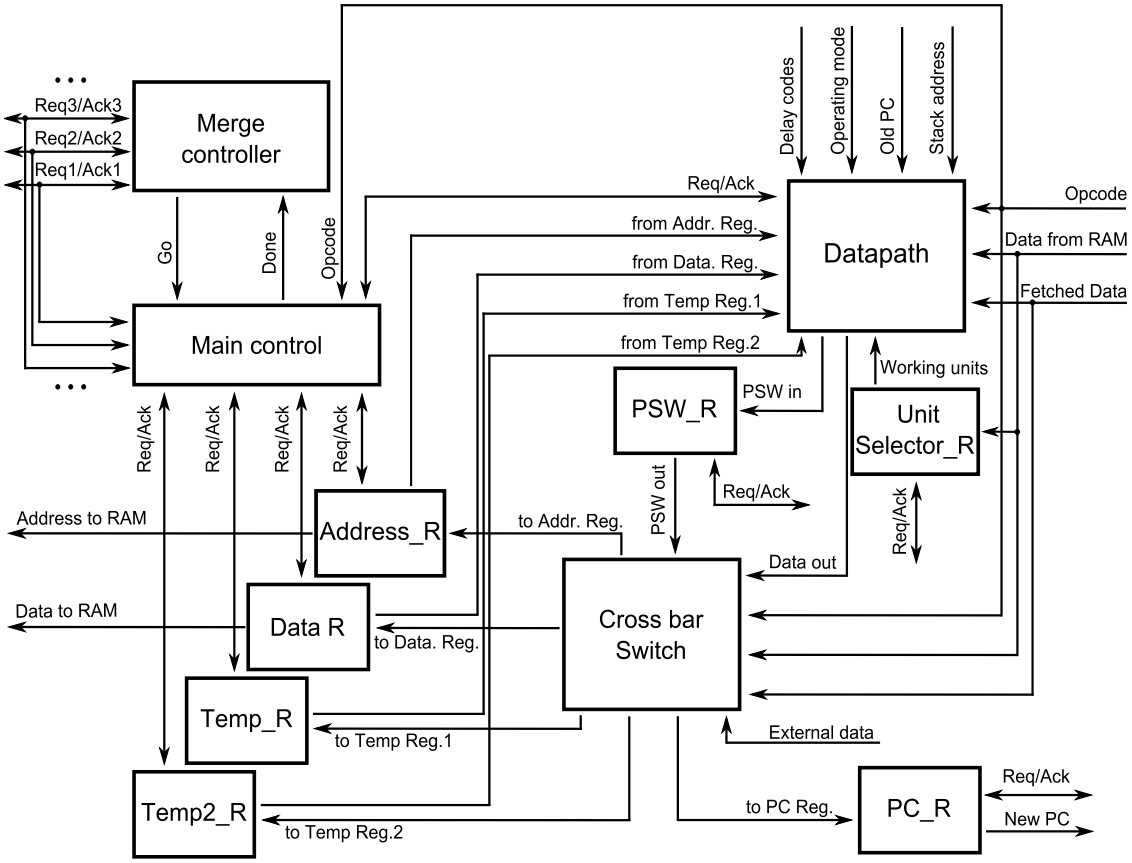


Figure 4.9: ALU internal structure

der) have a duplicate set of components to support multiple operation modes (Section 4.4.1) and fault tolerance features (Section 4.4.3). The block has "Operating mode" and "Working units" inputs, which provide information about the current operating mode (high performance/low power) and health of the computational units. There are other inputs and outputs, such as the "Delay codes" input (the delay code is provided for each arithmetic component and register with an adjustable delay line, see Section 4.4.2), the "Old PC" input (the current instruction address, which is used in branch instructions), the "Opcode" input (the type of operation to be performed and the input data for it – this information is encoded in the opcode of the operation), the "Data from RAM" and "Data from ROM" inputs, the "PSW in" output (new data to be written to the *PSW Register*) and the "Data out" output (data computed by the arithmetic blocks, which goes through the *Cross bar switch* to its destination).

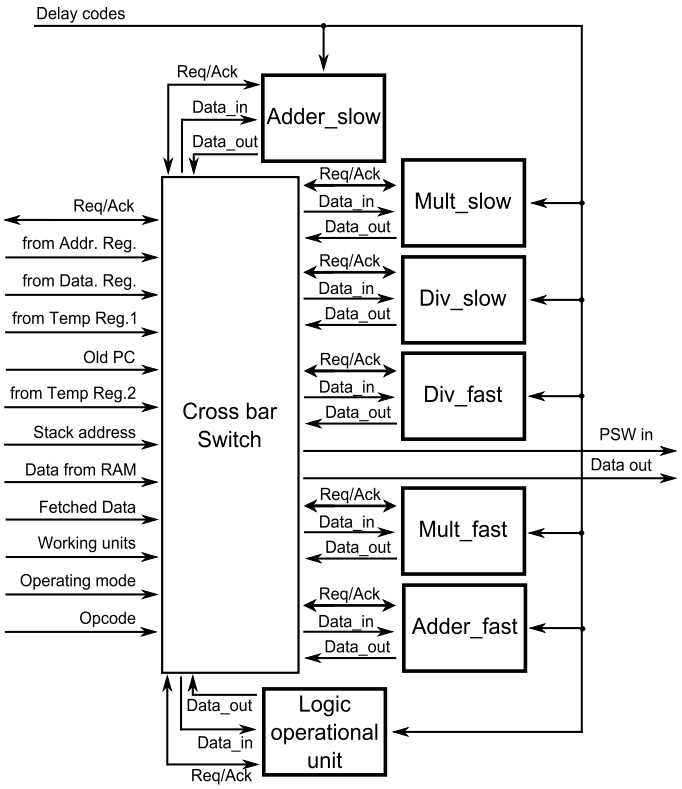


Figure 4.10: Datapath block internal structure

- *Cross bar switch* is an interconnect circuitry matching inputs and outputs depending on the "Opcode" input.

- *PSW* (Program Status Word) *Register* is a 16-bit directly addressable register, that holds information about the current CPU state. The structure of the register is shown in Table 4.4.

- *Unit Selector Register* is a 7-bit directly addressable register, which is used to hold information about faults in datapath components, each represented by a register bit (6 bits in total, see Table 4.5), the 7th bit ("*bulb*") is used for demonstration purposes only (it is accessible from the software level and connected to the output pin of the chip, see Appendix D). The register can be accessed by a special instruction *MOV wrk, direct*, which writes data from an internal RAM location into the *"Work" Register* (this instruction has been added to the standard Intel 8051 ISA).

Table 4.4: Structure of the PSW register

Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OV	*EA*	*N*	*Z*	*RS1*	*RS0*	*CY*

0. *CY* Carry flag is set if there was a carry from or borrow to the most significant bit in the last arithmetic operation.
1-2. *RS1* & *RS0* Register bank select: 00 – Bank0; 01 – Bank1; 10 – Bank2; 11 – Bank3.
3. *Z* Zero flag is set if the last arithmetic result is equal to zero, and reset otherwise.
4. *N* Negative flag is set if the last arithmetic result is negative, and reset otherwise.
5. *EA* Interruption flag indicates the occurrence of an interrupt.
6. *OV* Overflow flag is set a signed arithmetic operation result is too large positive or negative number to fit into the destination register.

The rest 9 bits (15..7) can be used as general purpose flags.

Table 4.5: Structure of the "Work" register

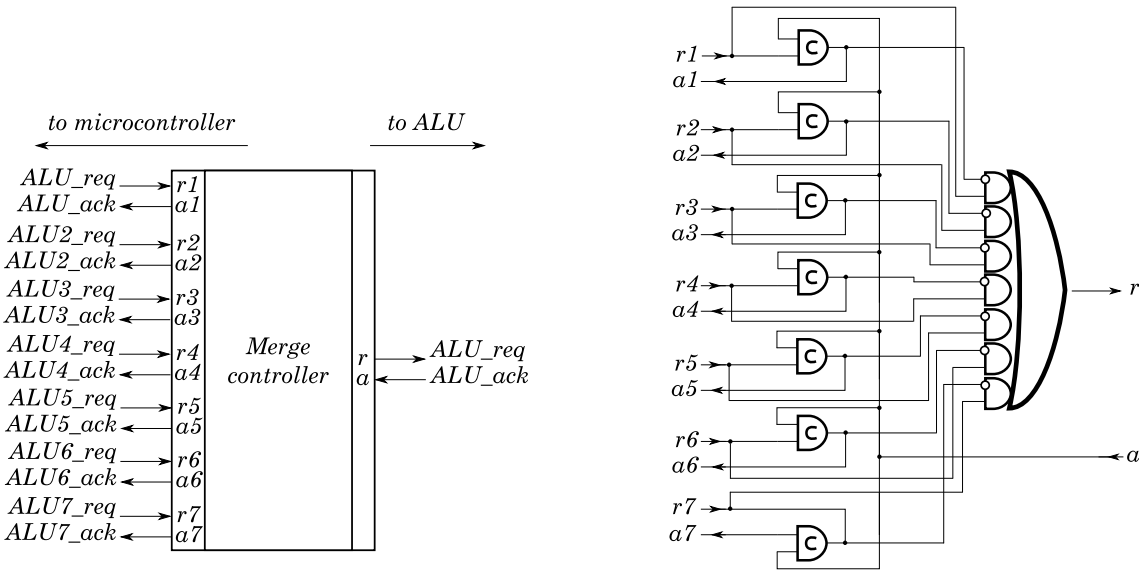
Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
"bulb"	*div1*	*div0*	*mult1*	*mult0*	*add1*	*add0*

0-5. – 6 bits carry information about fault in datapath arithmetic units: 0 and 1 (*add0* and *add1*) – adders; 2 and 3 (*mult0* and *mult1*) – multipliers; 4 and 5 (*div0* and *div1*) – dividers (see Section 4.4.1).
6. *"bulb"* – this bit used for demonstration purposes only.

- *PC* (Program Counter) *Register* is an an auxiliary 16-bit register holding a new PC address before it is sent to the Program Counter Incremental Unit.

- *Merge controller* is a special component that allows requests to the ALU multiple times during the execution of the same instruction. The ALU can be requested up to 7 times within the same scenario (see Figure 4.6) and ALU control logic needs to behave differently for each request. The merge controller was originally proposed in [102] to solve handshake management issues. We developed the implementation further to cope with a higher number of requests. Figure 4.11 shows a schematic view (a) and implementation (b) of the *Merge controller* used in the ALU component. The complex gate in Figure 4.11(b) was separately implemented and validated for timing hazards.

It is worth mentioning the schematic implementation of the merge controller for the ALU block (Figure 4.11(b)) was shown for the reader to understand the functional structure of the unit. During the implementation this block was described using VHDL and then generated by the synthesis tool. The actual structure of the merge controller block is different and contains several various complex gates generated by the tool.



(a) Schematic view (b) Implementation

Figure 4.11: Handshakes merge controller

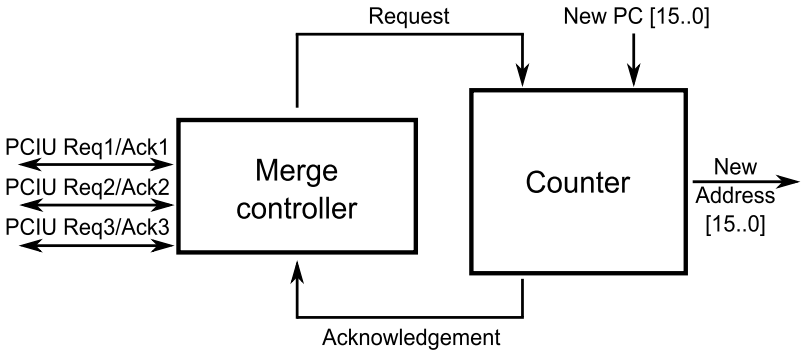


Figure 4.12: Top-level structure of the PCIU

All communication between the control logic and the datapath units within the ALU component (Figure 4.9) is arranged by means of request and acknowledgement signals (in the same way as the Top level, see Figure 4.2) and is regulated by the *Main control* block (see Section 4.2.2).

4.3.2 Program Counter Increment Unit (PCIU)

Each Von-Neumann processor contains a component, which stores the address of the instruction that is currently being executed. In the Intel x86 architecture such a unit is called Instruction Pointer (IP) [160] or Instruction Address Register (IAR) [109]. In our implementation it is a Program Counter Increment Unit (PCIU).

It is a 16-bit loadable counter, which is incremented before fetching new data from program memory (Section 4.3.3). Instructions are normally fetched sequentially from ROM, however, there are situations when a new value is placed in the PC, e.g. branching or jump instructions, calls and returns, etc. In this case a new value is prepared by the ALU block and loaded into the counter in the PCIU.

Figure 4.12 shows a schematic view of PCIU, which contains a 3-way Merge controller (similar to the one in the ALU, however the PCIU block can be requested a maximum of three times in the same instruction) and loadable 16-bit counter, which provides the address for the next data to be fetched from program memory.

4.3.3 Instruction Fetch Unit (IFU)

We followed the original Harvard architecture, where instruction memory and data memory are physically separate. Hence all instructions are stored in program memory (a programmable *ROM*). To access the memory and load instructions the following blocks were implemented:

- *Instruction Fetch Unit* controls the request/acknowledgement protocol between the main control block and the ROM (mainly by the use of Merge controller see Section 4.3.1), provides the instruction address from the PCIU to the ROM and receives new data from the ROM.

- *ROM* stores the program code for the microprocessor. This is an off-chip 128Kb Erasable Programmable Read-Only Memory (EPROM) with 16-bit address bus and 16-bit output data bus.

- *Instruction Register* is an internal 16-bit register, which holds the opcode for the instruction, which is currently executed . This register is latched when it receives the *Done_F* (see Figure 4.2) signal from the main control logic, which indicates the end of execution of the current instruction (see Figure 4.6). When a new opcode is loaded into the register, the *Finish* signal initiates execution of the new instruction.

Programs may contain not only one-word instructions (just an opcode), but also two-word and tree-word instructions. The second and the third words (the operands) are used differently depending on the addressing mode (see Section 2.3.3).

A multi-word instruction can be sent not only to the *Instruction Register* as the opcode of the instruction, but also to the ALU block as specific data for computation.

Delay Codes are also read from the ROM (see Figure 4.2). These codes are required by the adjustable delay lines (see Section 4.4.2) and stored in the Delay Registers (DR) (see Section 4.3.6). During the Reset stage of the CPU, the IFU block generates the ROM addresses, which depends on the current value of the Delay Bit. If the bit is not set, the requested addresses would be from #0000h to #000Fh, if it is set – from #0010h

to #001Fh. Every value read from the ROM will be written to the corresponding DR (see Section 4.3.6).

4.3.4 Memory Access Unit (MAU)

In the previous Section we discussed the process of fetching instructions from ROM, however general purpose data is stored in a physically different unit called RAM. A special block, called Memory Access Unit, was designed to access this memory.

According to the original architecture [158] there are two RAM blocks (Internal and External) and different instructions are used to access each of them. Due to the lack of off-chip pins the External RAM was also placed on-chip and has the same size as the Internal one (512 bytes). Depending on the instruction (opcode), MAU accesses the required memory block and proceeds with a write or read operation. During the read operation the ALU calculates the address and sends it to the appropriate RAM through the MAU block. The data from the memory is send back to ALU directly. The same procedure is followed during the write operation, however along with the write address the ALU also needs to provide the data to be written.

MAU has its own Merge controller (Sections 4.3.1; 4.3.2) to deal with multiple requests from the main control logic, as it could be requested up to six times during the execution of the same instruction.

The Delay Code input provides the delay constraints for the adjustable delay line (see Section 4.4.2).

4.3.5 Stack Increment/Decrement Unit (SIDU)

The microprocessor has a special address space in RAM reserved for *Stack*. This area is allocated to store information about the current program status (PSW), specific registers (A, B [158]) or any other data, which can be corrupted during the interrupt handing process.

To have access to a memory location in *Stack* we need to store its address. For this reason a specific block, called Stack Increment/Decrement Unit, was designed, which

stores the address and provides functionality for incrementing the Stack Pointer (SP) every time we write to the Stack, and decrementing it when we read the data back. It has the same structure as the PCIU block (see Section 4.3.2), however the counter can both increment and decrement its value, and can't be loaded like PCIU. SIDU can only be requested twice in the same instruction, so it uses a 2-way Merge controller.

4.3.6 Delay Registers (DR)

Delay Registers (DR) is a set of 8 internal 16-bit registers, to store the Delay codes for adjustable delay lines (see Section 4.4.2).

The latency of a delay line is controlled by the value of the corresponding delay code, so the valid code needs to be stored in the DR. The process of loading codes into the registers is controlled by the IFU block (see Section 4.3.3). Table 4.6 shows the allocation of the Delay Registers to specific computational unit delay lines.

Every time the environment conditions (voltage supply, temperature, etc.) change, the critical path of the computational logic is affected, so we need to update the Delay code. In order to do so, the CPU needs to go through the Reset process, when the IFU block (see Section 4.3.3) generates addresses (depending on the Delay Bit), fetches a new Delay codes, and loads them into the DR.

In out implementation the only way to load new constants is to go through the reset process, so it might seems that there is no seamless changes with the environment. However this is the way how we needed to deal with the bundled data approach. It is our future work to apply other approaches, e.g a competition detection techniques, to remove the this issue completely.

4.3.7 Interrupt handler

Following the original CPU implementation, our microprocessor is able to handle hardware interrupts.

Usually a hardware interrupt is a special signal sent to the processor from external devices in order to execute a particular routine and pause the running program. The

Table 4.6: Structure of the Delay Registers set

Delay Register	Bits of the Delay Code	Datapath Component
1	[15...8]	Fast Adder
	[7...0]	Low-power Adder
2	[15...8]	Fast Multiplier
	[7...0]	Low-power Multiplier
3	[15...8]	Fast Divider
	[7...0]	Low-power Divider
4	[15...8]	Internal RAM
	[7...0]	External RAM
5	[15...8]	ROM
	[7...4]	Program counter (counting phase)
	[3...0]	Program counter (loading phase)
6	[15...12]	SIDU register
	[11...8]	PSW register in ALU
	[7...4]	Address register in ALU
	[3...0]	Data register in ALU
7	[15...12]	Temp register in ALU
	[11...8]	Temp register 2 in ALU
	[7...4]	"Work" register in ALU
	[3...0]	Logic operations in ALU
8	[15...8]	Instruction Register

control logic of the CPU needs to have the ability to recognise such a signal and initiate a particular process to handle the interrupt using appropriate datapath units.

Due to the fact that the original Intel 8051 processor was specifically designed to work as a microcontroller, the main CPU was accompanied with a number of peripherals, such as counters, watchdog timers, ADC/DAC converters, etc. Hence, it was supplied with a variety of interrupt levels. However, our goal is to design a CPU with specific features, so a less sophisticated interrupt handler was implemented.

We have implemented the *interruption* bit, which indicates the occurrence of an interrupt. It can be found along with its *opcode* in Figure 4.6. Figure 4.13 shows a graph describing the order of activation of functional units in the situation when a processor interrupt occurs. This graph can also be found on the right hand side of the CPOG representation of the complete instruction set. Execution of every instruction finishes in vertex *DONE*, no matter whether interrupt happened or not. If there were no interrupts during the execution of a PO (i.e., the interrupt bit was not set (see Figure 4.7(b)), the execution continues to vertex *DONE_F*, representing the end of the instruction execution. However, if an interrupt did occur (we assume that the processor wasn't in the state of handling a previous interrupt) then the execution would follow a different route (Figure 4.7(a)). First we need to save the current program counter (*PC*) onto the stack ($ALU/6 \rightarrow MAU/6$) so that the processor can return to the location where it was interrupted, and set a special "Interruption flag" (*EA*) in the PSW register (Figure 4.3.1) to indicate the occurrence of an interrupt. Then we update the stack pointer (*SIDU*) concurrently with loading the interrupt handler address (the entrance to the interrupt handler subroutine has address *FF00*) into *IFU* ($ALU/6 \rightarrow MAU/6$). Finally, the execution finishes in vertex *DONE_F*. Note that vertices *GO*, *DONE* and *DONE_F* do not represent any functional units (unlike other vertices); instead, they indicate the start of an instruction execution or its completion. At the end of the interrupt handling procedure the programmer needs to disable the *EA* flag (by *RETI* instruction), so that a new interrupt can be handled.

If a new interrupt takes place, while the processor is handling the previous one, we ignore it until the "Interruption flag" (Figure 4.3.1) is reset, indicating that the CPU is

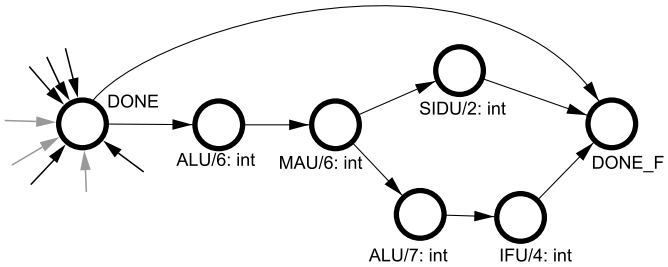


Figure 4.13: PO representation of interrupt handling

ready to process the next one.

4.3.8 Communication protocol between control and datapath units

Digital circuits register the computation results when the operation completion signal is issued. In synchronous circuits the role of such a signal belongs to the global clock whose period is chosen to be long enough for all the circuit modules to complete the computation, thus exhibiting the worst case performance. However, as the title of the Chapter suggests we designed an asynchronous microprocessor, where it is achieved by allowing each module to indicate its progress independently, either through explicit completion detection logic or by replicating the critical path in the form of a matching delay line [139]. The former approach requires redesign of the datapath components (using, e.g., *dual-rail logic*) with associated design overheads and productivity penalties. The latter approach, called *bundled-data*, allows the reuse of conventional design methods and existing datapath components, and thus is more convenient for our purposes.

Two signalling disciplines can be exercised over a bundled-data channel – 2-phase and 4-phase. A 2-phase protocol indicates the availability of results by any change of the completion signal, which requires a more complicated control logic. In a 4-phase protocol only the rising edge indicates the availability of the results, which simplifies the control logic, but introduces latency overheads because of the mandatory reset phase. However, these control delay overheads can be efficiently mitigated by using asymmetric delay lines [51] or local clock controllers [116], therefore we chose 4-phase signalling for our design.

In the next Section we will focus on the new features which were introduced in the

CPU for the purpose of power proportionality, fault tolerance, a wide range of operating modes, design for test issues, etc.

4.4 Optimisations

In the Introduction (see Section 1.1) we mentioned that there is a high demand for system, which can operate in a wide range of supply voltages, and adjust their functionality towards a specific application and operating mode. In this section we will discuss these issues and introduce several important features of our design: Extended datapath structure (Section 4.4.1), Adjustable delay lines (Section 4.3.6) and Fault tolerance mechanisms (Section 4.4.3).

4.4.1 Proposed extended microprocessor datapath

As discussed in the introduction, it is important to provide support not only for dynamic reconfigurability in the application-specific context but also to capture multiple operation modes provided by the system. Such reconfigurability can be applied at different levels of abstraction – from the high level of system components down to individual gates and transistors. We focus on the functional block level of granularity.

The most power and time consuming components of the 8051 microprocessor are in the datapath [151], e.g. adders, multipliers and dividers. To be adjustable towards a wide range of operating conditions and custom applications, we designed two sets of arithmetic units: one optimised for energy consumption and the other one for performance:

Adder implementations. As we needed to have two adders different in their performance and power consumption, we looked through several existing implementations: Ripple Carry Adder (RCA) [29], Carry Look-ahead Adder (CLA) [77], Kogge-Stone Adder (KSA) [87] and Brent-Kung Adder (BKA) [25]. Amongst these four implementations, RCA and CLA are slow, but low-power, however KSA and BKA or so-called Prefix-Tree adders [25] target high-performance applications. After comparing the simulation results, we chose RCA and BKA, see Table 4.7.

Multiplier implementations. A similar search was done to find two examples of mul-

Table 4.7: Comparisons between different implementations of arithmetic logic

Computational unit	Type	Delay (ns)	Power consumption (W)	Energy per operation (J)	Area (units)
Adder	fast	1.81	2.93e-06	2.95e-08	4687
	slow	2.01	1.67e-06	2.44e-08	4113
Multiplier	fast	2.25	3.76e-05	1.71e-07	20970
	slow	4.13	2.78e-05	1.49e-07	14756
Divider	fast	18.61	1.14e-05	5.76e-07	44275
	slow	22.09	1.07e-05	5.58e-07	43576

tiplier units. There are several multiplication algorithms that are used nowadays, such as sequential multipliers [141], Booth's multipliers [24], Wallace [154]/Dadda [52] tree algorithms, etc. Moreover, most implementations come with various modifications of adders and encodings, depending on which the performance and power consumption of the final multiplier may vary. For high performance multiplication we chose Wallace tree design and for low power we chose a simple partial products multiplier (PPM) [77]. Simulation results are shown in Table 4.7.

Divider implementations. We tested several designs of divider blocks (Long division algorithm [157], Nonrestoring divider [156], SRT division [127], Goldschmidt Implementation [63], etc.) to find appropriate examples of high performance and low power divider logic. High performance divider was taken from the DesignWare Synopsys Library [73] and the "Long division algorithm" was chosen for the low power implementation. Simulation results of the chosen dividers can be found in Table 4.7.

Depending on whether there is a shortage of energy or on any other restrictions imposed by a custom application, we can choose the most appropriate functional block to be used during an instruction execution by switching a specially dedicated pin on the chip (see Appendix D) – the *Operating mode* bit (Figure 4.2). The decision on which set of units to use can be made at different levels: software, sensors, external signals, etc.

In the next version of our chip we are planning to add a power gating mechanism to switch off the unused set of computation components and thereby reduce the static

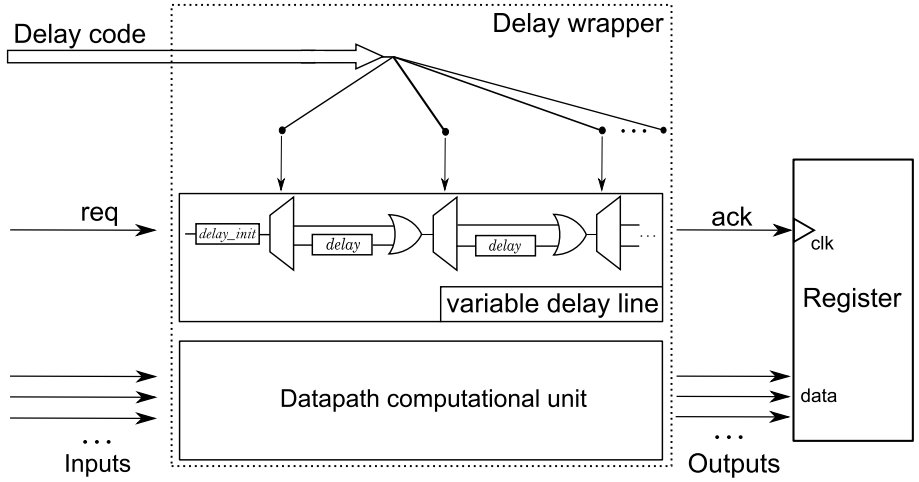


Figure 4.14: Configurable datapath component with adjustable delay line

power consumption.

Duplication of the main arithmetic units provided not only the ability to adapt to varying application requirements and operating conditions, but also allowed us to provide fault tolerance mechanisms (Section 4.4.3).

4.4.2 Proposed adjustable delay lines

Our implementation of self-timed datapath components is based on the bundled-data approach (Section 4.3.8), where each computational block (Table 4.6) is accompanied by a matching delay line to signal completion. In order to correctly function in a wide range of operating conditions (e.g. supply Vdd or temperature), the bundle-data component needs to adjust the latency of its completion signal. We propose to address this issue by use of an adjustable delay line [129], whose latency is selectable by a *Delay code*, see Figure 4.14.

The *ack* signal is produced as a reaction to the *req* signal after a time interval which is chosen by the *Delay code* input - the total latency is formed as a combination of delay portions between the multiplexers controlled by the *Delay code* bits.

The *Delay code* was calculated according to Vdd, environment temperature and technology process values. First we synthesised a netlist for a computational unit (such as an adder or a register) using the chosen technology library (130nm CMOS, see Section 5.1).

Then the netlist was simulated under normal corner case conditions (nominalV_{dd} level 1.2V) in order to find out the critical path delay of the component. This value (minus the delay of all the OR-gates, multiplexers and wires in the delay line) represents the time delay in the *delay_init* (Figure 4.14) block. Hence under normal operating conditions when all the bits in the *Delay code* are "0", after the *delay_init* time the *ack* signal will represent the completion of calculation of the computational unit.

When the operation conditions (V_{dd} and/or temperature) change we need to add extra delay portions into the delay line to generate a valid completion signal. By knowing the number of bits in the *Delay code* for a particular variable delay line (Table 4.6) and the dependency of the propagation delay and the operation conditions, we split the delay portions equally between the *delay* (Figure 4.14) blocks in each of the delay line. By applying a particular *Delay code* we can adjust the completion signal as widely as possible.

The *Delay codes* are loaded into the *Delay Register* (Section 4.3.6) during the reset stage of the microprocessor using the IFU block (Section 4.3.3).

4.4.3 Fault tolerance

All complex systems, including microprocessors, are designed with the possibility of faults in mind. Sometimes it may be impossible to predict the nature of potential faults and their locations. It is therefore important to design a system in such a way that it can tolerate the faults and continue functioning correctly. A general approach to building fault tolerant systems is redundancy, which can be applied at several levels:

- time redundancy - by performing an operation several times;
- data redundancy - by providing extra information;
- physical redundancy - by supplying extra hardware to allow the system to compensate the loss of failed components [122].

As our design has a duplicate set of computation blocks (Section 4.4.1) for the ability to work in multiple operation modes, we can make use of physical redundancy and build a

fault tolerant system. For this a special *"Work" Register* (see Figure 4.9) is provided. It holds information about faults in datapath components, each represented by a register bit (6 bits in total) (Table 4.5). The register can be accessed by a special instruction ***MOV wrk, direct*** – write data from an internal RAM location into the *"Work" Register* (this instruction has been added to the standard Intel 8051 ISA). Each arithmetic component can be checked for operational correctness (by performing arithmetic operations on them) and if one of the components is not working properly, the *"Work" Register* can be updated accordingly.

It is important to note that functional correctness has a higher priority than the operating mode. In other words, if a particular datapath component is chosen for a specific operating mode, but the "Work" Register states it is broken, then a duplicate component will be used instead. This choice is done hardware level, however the "Work" Register can be changed by the operating system.

4.5 Design for test

In Section 4.4.3 we discussed how the system tackles faults within the computational blocks (adder, multiplier, etc.). However, faults can occur in various parts of the system and it is more important to locate such defects even before the design starts functioning. In this Section we address these issues by using Design for Test (DFT) techniques, which is a set of design methods that add testability features to a design and validate the system for a functional correctness after it has been manufactured.

The choice of a particular testing methodology largely depends on the nature of the fault. Section 4.5.1 gives a quick overview of a variety of errors that can occur in the process of a chip development, different types of fault models that are used to describe such defects and a range of testing methods applied to detect a particular error.

In Section 4.5.2 we discuss particular DFT techniques used in our implementation.

4.5.1 The fault types and DFT techniques

In the process of a VLSI circuit development a number of error/fault models are used:

- Logic errors. This group includes all the functional errors made during the design or fabrication stages of development. Different fault models are used for this kind of defects, the main types of these are: Single Stuck-at (when one of the nodes in the circuit steadily tied to either logic 0 or 1), Stuck-Open model (when a physical line in the circuit is broken and tied neither to 0 nor to 1) and Bridging (two or mode nodes of the circuit are shorted together).

- Delay faults. Some physical errors, i.e. process variations, make some delays in the circuit greater or smaller than expected. Typically two fault models are used: Transition fault model (or gate delay) and Path delay fault model.

- Current-based model (IDDQ fault model [126]). This method suggests measuring the steady state current of the device against a predefined pass/fail threshold.

The probability of occurrence of a particular type fault depends on the technology used. If the technology is $\geqslant$ 130nm, then the above models usually cover 90% of all possible fabrication defects. However, if we go below that technology, such models have a low fault coverage and people use other less-popular models, such as a low-stress voltage faults models (usually low Vdd increases the delay of the circuit, more so for a faulty one; tests on higher Vdd can reduce the lifetime), power monitoring (static/dynamic IR drop during the test [110]), etc.

For detection of a particular fault it is possible to generate a specific test pattern by Automatic Test Pattern Generator (ATPG) tools [47] using various algorithms (Pseudo-random, Ad-Hoc, PODEM, etc.).

The most conventional way to gain test information from the circuit under test (CUT) is to use so-called *scan-chains* [159]. This is a technique which provides a simple way to set and observe all the registers of the design. Test patterns are shifted by *scan-chains* into registers, then the clock signal is issued to test the CUT, and the results can be shifted out by *scan-chains* to be compared with the expected results.

We use the *scan-chain* approach in our implementation (Section 4.5.2).

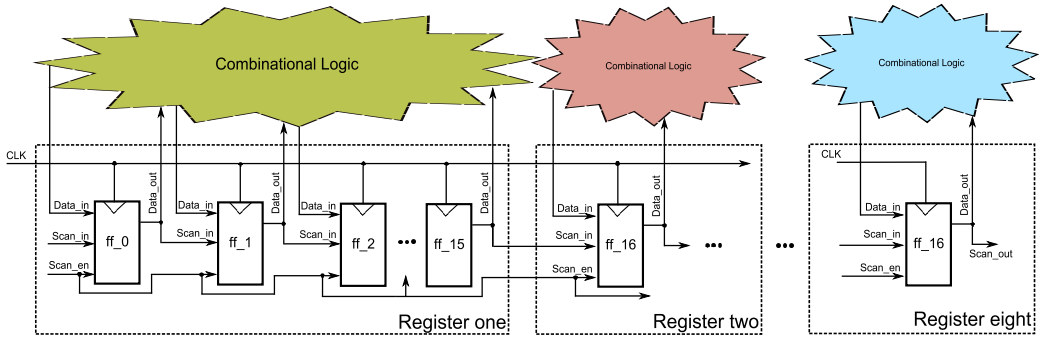


Figure 4.15: Implementation of the Delay registers using the *Scan-chain* technique

4.5.2 DFT techniques

Section 4.5.1 gave a quick overview of the main DFT methods. However, depending on the design targets, those approaches can be used to a different extent, such as full scan, partial scan, boundary scan, etc. To save area and design time, we implemented a partial scan for our design, hence only dedicated flip-flops were chained.

It is crucial for our design to have the Delay registers (Section 4.3.6) properly loaded with the Delay codes, as without them the completion signal from the adjustable delay lines (Section 4.4.2) will be incorrect and therefore the results from the corresponding computational logic will be registered wrongly. A *scan-chain* of eight 16-bit registers from the Delay register set was implemented, see Figure 4.15.

The selected flip-flops (FF) from the design were replaced by scan-FFs (using a *DFT Compiler tool* from Synopsis [143]), which contain several additional inputs. In *test mode*, when the Scan_en input is set, all FFs are configured as a chain of shift registers. The test pattern is then shifted in using Scan_in and CLK inputs. Once the whole pattern is shifted in, the Scan_en is reset and the Delay registers are back to the *normal mode* to apply the pattern on the connected combinational logic. Once this is done the circuit is put back to *test mode* and results can be scanned out from the registers using Scan_out.

Some implementation details and results regarding the testing techniques used in our Demonstration chip are presented in Section 5.3.

4.6 Conclusions

This chapter described the main stages of our design flow and provided implementation details of our asynchronous 8051 microprocessor. We outlined a novel CPU design methodology (Figure 4.1), which was followed through the whole design process:

- Using the provided CPU specification, we analysed the processor architecture and instruction set (Section 4.1).
- Section 4.2 described the process of the control logic development and highlighted the benefits of using a novel formalism of CPOGs.
- The structure of the datapath was outlined in Section 4.3. We discussed the main features of our implementation: extended datapath structure (Section 4.4.1), adjustable delay lines (Section 4.4.2), fault tolerance mechanisms (Section 4.4.3) and DFT methods (Section 4.5).

In this chapter we discussed the design techniques we used to develop a system which can operate in a wide range of supply voltages and can adjust its functionality towards a specific application and operating modes. Such a system can adapt itself depending on the energy budget and computational resources availability, so it contributes to the so-called power-proportionality criterion [10].

The next chapter demonstrates the feasibility of our approach by building a competitive asynchronous microprocessor, and presents a demonstration in silicon, which was produced during this PhD work.

Chapter 5

Implementation of the Asynchronous 8051 microprocessor demonstrator chip

In the previous Chapter we discussed the architecture and the main design features of our new Asynchronous 8051 microprocessor. To demonstrate the feasibility of our methodology, novel design flow, and optimisation techniques a demonstrator silicon was produced. This chapter addresses the CPU design flow, which was discussed in Chapter 4, in terms of hardware development and synthesis for the chip fabrication. Section 5.2 discusses the implementation of the microprocessor's control logics (both the Top-level and the ALU control logic) and the datapath synthesis is described in Section 5.3. After all the main parts of the chip were synthesised and tested, they were integrated into the whole chip design and enriched with DFT features for offline testing. The resultant design was simulated, verified and finally fabricated (Section 5.4). Once the ASIC was received from the manufacturer, the evaluation procedure took place. To validate the functionality of the microprocessor we developed a testing board, whose details are shown in Section 5.5. All the measurements, analysis and comparisons with synchronous and other low-power CPU implementations are presented in Section 5.6.

5.1 Introduction

The history of the design of an asynchronous Intel 8051 using the CPOG methodology goes back to July 2011, when for the very first time its simplified FPGA implementation and measurement results were presented [132]. It took about a month to build that version which had a minimal instruction set capable of executing a simple program. The design was implemented and measured using two different FPGA chips (Flex10K [13] and Cyclone III [12]). We showed performance, power and area utilization advantages compared to its synchronous counterpart as well as to its asynchronous version implemented using the Balsa language [16, 2]. It was then decided to implement the entire Intel 8051 microprocessor with the complete instruction set. Several additional optimisations and improvements were considered during the chip design stage, as discussed in Chapter 4.

As the first try was purely an FPGA implementation we needed to go through a very complex design flow to fabricate the design in silicon. This flow includes the following important stages:

1. Behavioural description of each module and its validation by simulation.
2. Register-transfer-level (RTL) implementation, its simulation and verification.
3. Validation using an FPGA development board.
4. Hardware synthesis for a particular technology library.
5. A number of validation procedures (such as simulation, static timing analysis, etc.) to check the functionality of the design before proceeding with layout.
6. Place and Route (P&R) and physical validation.
7. Parasitic Capacitances Extraction and post-layout simulations.
8. Sign-off and fabrication.

Figure 5.1 shows a diagram of the explained stages and their interconnections.

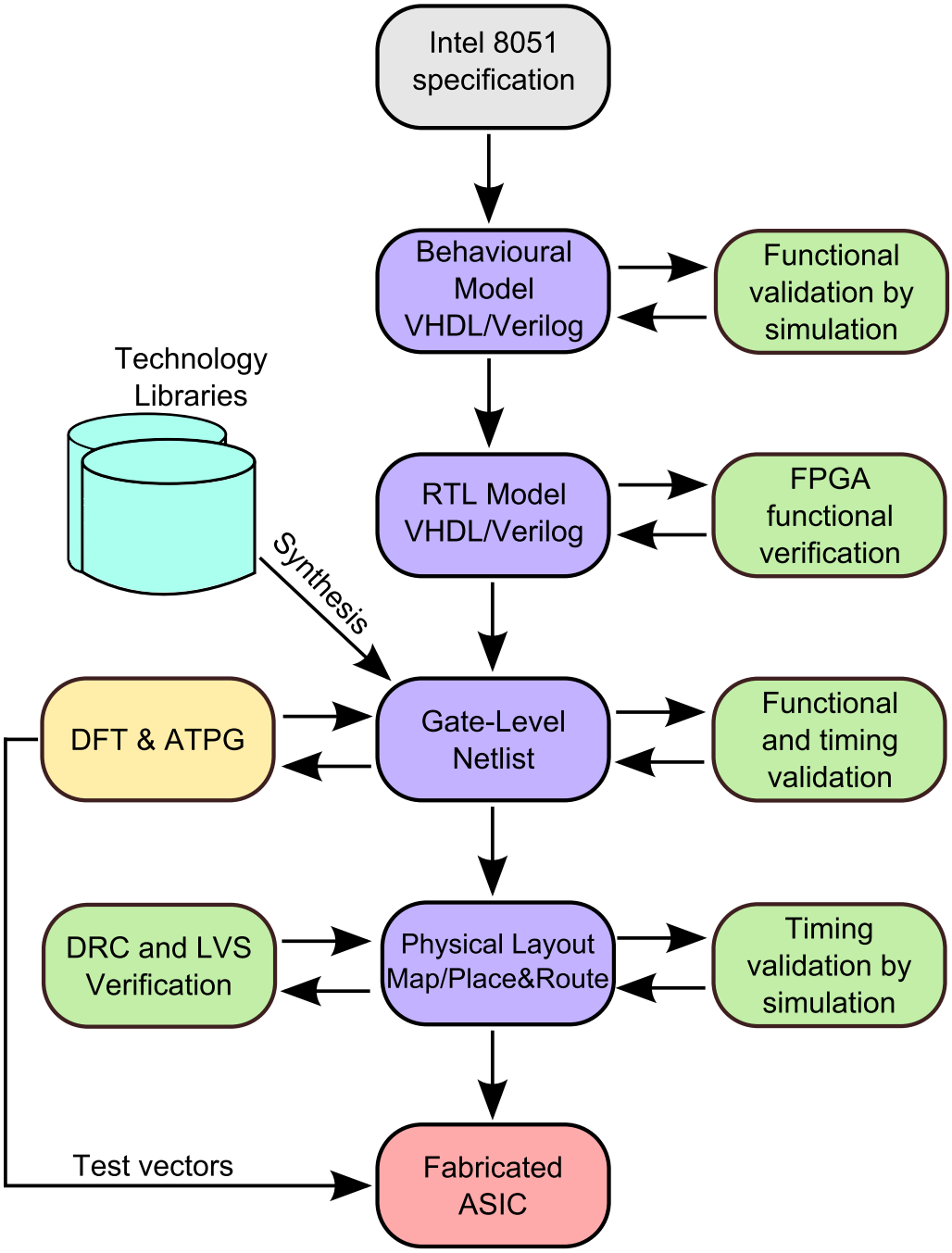


Figure 5.1: Stages of the design flow

To successfully accomplish all these stages we employed various Electronic Design Automation (EDA) tools provided by different companies: Synopsys, Inc. tools for behaviour/RTL synthesis and simulation such as Design Compiler [142], Verilog Compiler Simulator (VCS) and Discovery Visual Environment (DVE) [153], PrimeTime suite [147] for

power and time measurements; FPGA design tools such as Altera Quartus II design [53] and Altera FPGA developing boards (DE0 and DE1 Altera Developing FPGA board [12]); Cadence Design Systems, Inc. tools for layout such as Encounter(R) [30] and Mentor Graphics tools for physical verification such as Calibre [99].

It took about 5 months for the design to go through the above stages. The control logic was the easiest part to design, since it was benefiting from the presented compositional approach. We reuse part of the previously developed control logic from a simplified version of 8051 microprocessor [131]. However the most time consuming part to design was the processor's datapath, as it contained more blocks to be developed and validated. Moreover since we were implementing a self-timed design, the datapath has an asynchronous nature, but it is a well-know fact that currently the development of asynchronous circuits has been hindered by the difficulties to design self-timed systems using existing EDA tools. In this aspect extra time was needed to verify the correctness of these circuits.

The chip was meant to be a "proof-of-concept" of the feasibility of the CPOG approach and a demonstration of power-proportionality as a method of building energy-efficient and adaptive systems (see Chapter 1).

The next section will focus on the implementation details of each of the microprocessor's internal blocks presented in the entire architecture for the CPU (Figure 4.2).

5.2 Control logic implementation

As it was mentioned in the introduction, the control logic design benefited from using the novel CPOG formalism, which allowed us in a convenient way to extend our previous implementation [132] of 4 instructions to current 257 instructions ISA.

In Section 4.2 we presented the specification and the complete design flow of the Top-level and ALU control logics, so in this section we explain their hardware implementation and verification details.

5.2.1 Implementation of the Top-level and ALU control logics

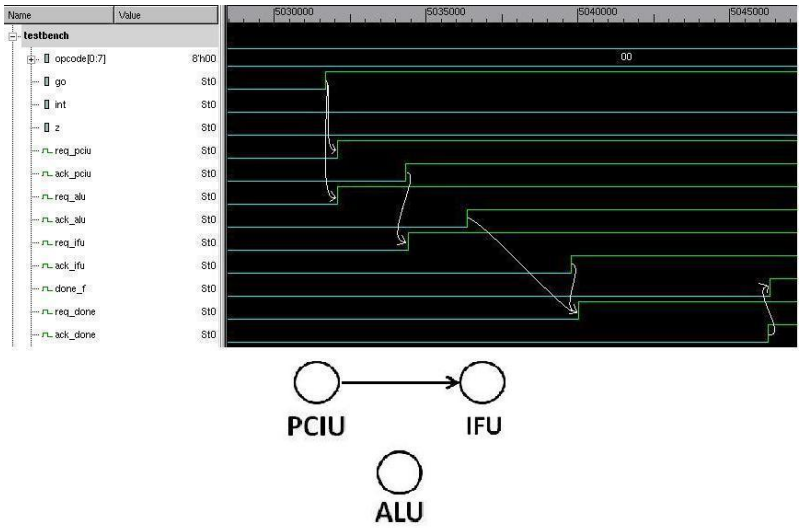
As it was mentioned in Section 4.2 the final stage of the control logic design is the mapping of the synthesised CPOG into Boolean equations[1], which were then translated to VHDL code. After generating VHDL files we, following the flow from the Figure 5.1, first of all, verified the functionality of both control logics using an FPGA designing tool (simulation and development board) and then by simulating a synthesised Verilog netlist. Simulation of the Top-level control logic's netlist is presented in Figure 5.2(a), where we can see a correct execution of a PO (*class H* (see Appendix A.8)). After the start signal *GO* the PO execution begins according to its description (under the waveform) with two request signals (*req_pciu* and *req_alu*), then after an acknowledgement from the PCIU is received, a new request to IFU block (*req_ifu*) is generated. Finally when acknowledgements from both the ALU and the IFU blocks are received, the signal *done_f* was issued, representing the end of the PO and instruction.

Another example of PO simulation using an Altera FPGA design tool is presented in Figure 5.2(b). For clarity we do not show acknowledgement signals here, however the simulation is the same as the previous example. Figure 5.2(c) presents a sequential simulation of the previous two POs in an FPGA design tool. The same verification procedure was repeated for each PO class.

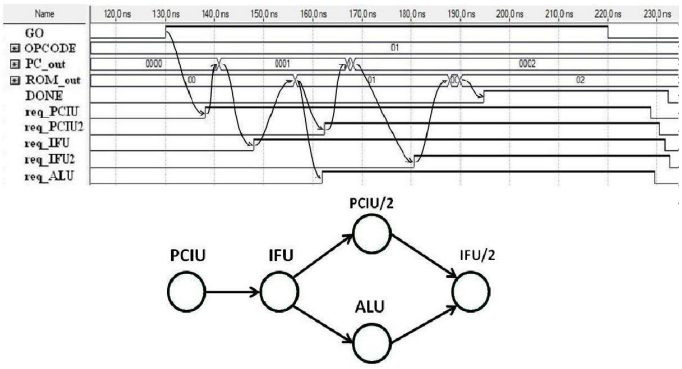
The ALU control logic was synthesised and verified for functional correctness in the same way as the Top-level one.

To estimate the complexity of the generated control logic, the number of cells used for the top-level control (326) and the internal ALU control (220) was counted. It should be noted that in the used technology a cell can correspond to a logic gate with up to 9 inputs. The total area for both of the microcontrollers (top-level and ALU logics) was only 546 logic gates. For comparison, we took three publicly available Intel 8051 implementations, namely [3], [4], and [5], and synthesised their central controllers in the same technology library. The final gate counts were, respectively: 1545, 472 (without the ALU/interrupt control), and 825. The ALU and interrupt control logic from [4] was scattered across

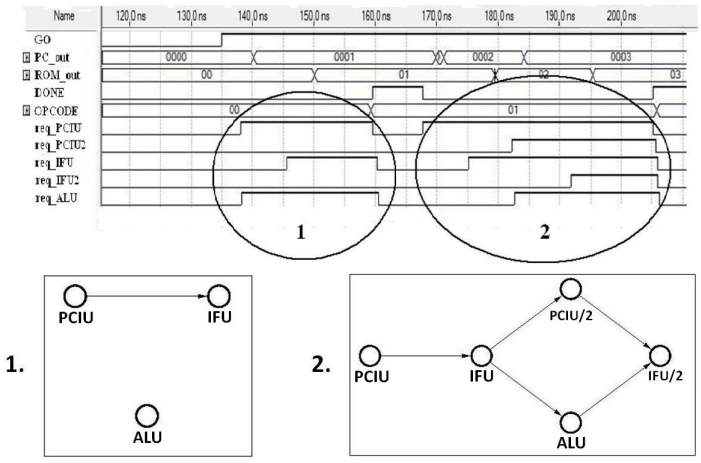
[1] All the resulting equations for both of the Top-level and ALU control logic blocks are shown in Appendix B.



(a) Simulation of PO execution using EDA simulation tool



(b) Simulation of PO execution using FPGA simulation tool



(c) Simulation of sequential execution of two POs

Figure 5.2: Waveform of example PO simulation

datapath modules for optimisation, hence we could not extract it and it was not included in the count of 472. However, we can still conclude that our implementation is efficient in terms of area.

After the control part was synthesised and checked we focused on implementation of the second main part of the CPU – the Datapath, which is discussed in the next Section.

5.3 Datapath implementation

In Section 4.3 we outlined details of functionality and the main features of the the Datapath, whose hardware implementation details are discussed in this section.

The synthesis of the processor's datapath followed its hierarchical structure presented in the overall architecture for the CPU (Figure 4.2): synthesis of the ALU block (Section 5.3.1), PCIU implementation is presented in Section 5.3.2 and finally the IFU component with the Delay Registers are explained in Section 5.3.3.

There are other important hardware components, whose implementation details are summarised below:

- **MAU**

  The main purpose for the MAU is accessing both of the RAM (internal and external) blocks (see Section 4.3.4), therefore it redirects the address and data from the ALU to a specific memory block depending on the opcode and received data from memories back to the ALU. The RTL component was synthesised according to the specification, verified and tested.

- **ROM and RAM blocks**

  These two units were selected from the list of memory configurations supported by the fabrication company. According to the power consumption, performance and area requirements we chose two 512 byte modules SPSMALL9gp_256X16m2 for both of the RAM blocks and SU180_65536X16X16BM4A for the ROM. However this ROM block was only used for simulations, as in our implementation we had an off-chip 128Kb EPROM (AT27C1024).

Both RAM components are located on the chip, so we needed to provide a facility to read them independently from the main microprocessor. For this a special component was developed to support two operating modes of the chip: the *test mode*, when some of the external I/O pins are used to read the RAM components and the *work mode* when the memories are used by the CPU. The selection of these modes is managed through special pins, which were added to the floorplan: the "mode_select" to switch between two modes and "ram_select" to switch between two RAMs (Figure 5.9(a)). The code explaining this procedure is given in the Appendix E.

- **SIDU**

  The main purpose of the SIDU block is to provide the stack address, hence we implemented a counter, which can be incremented or decremented depending on the opcode. Regarding the original architecture (see Section 2.3.2) the stack is located at a particular area in the internal RAM (30h – 7Fh), therefore the first stack address generated by the SIDU block will be 30h. A programmer needs to be aware of the stack pointer as the stack and user variable are sharing the same memory area.

- **"Design for test" implementation**

  DFT methods are well-used in the ASIC design as they significantly improve circuit testability after the chip was manufactured.

  For our implementation it was crucial to have the Delay registers with valid Delay codes, as without them the microprocessor will malfunction. For this purpose a special Scan chain of 120 FF was automatically generated through our 8 Delay registers using the DFT Compiler tool. This Scan chain was then simulated and verified.

  As we used a partial scan approach the total area overhead wasn't significant (less than 1% of the total chip area) compared to implementation without the DFT, however when the ASIC was fabricated the chain played a significant role in the

ASIC's validation and testing.

The DFT approach requires several extra I/O pin (e.g. scan_in, scan_en, scan_out, scan_clk and scan_mode) added to the floorplan of the chip. In this aspect we used the same "test mode" (as for reading RAM blocks) to scan in/out our test vectors in the scan chain. The code explaining this procedure is shown in Appendix E.

The rest of the Datapath implementation is described in following section.

5.3.1 Synthesis of the Arithmetic Logic Unit

The main functionality and structure of the ALU block was addressed in Section 4.3.1. Similar to the top-level hierarchy, the ALU block was divided into the control and datapath parts. Synthesis and verification of the control part followed the same pattern as the top-level control unit (Section 5.2.1). The datapath consisted of internal registers (*Address* and *Data*, *Temp* and *Temp2*, *Unit Selector* and *PC*) and a Datapath block, with all the main arithmetic and logic units in it.

The implementation of 16-bit internal registers was straight forward, however the Datapath block needed to be carefully developed and verified as each its arithmetic component (adders, multipliers and dividers) was implemented in two styles: one optimised for low energy consumption and the other one for high performance. Various implementations of arithmetic units were reviewed in the Section 4.4.1, eventually we took open-source implementations for our datapath, i.e adders (RCA [138] and BKA [135]), multipliers (Wallace tree [44] and PPM [137]) and dividers (Long division algorithm [136] and divider from the Synopsys library [73]). Simulation results of the chosen arithmetic components are presented in Table 4.7.

After both types of the units were synthesised and verified, we combined them into the Datapath block. A special "Operating mode" bit was added into the chip floorplan to choose a particular set of operational arithmetic blocks.

These computational units as well as other datapath components were implemented in an asynchronous way and were based on the bundle-data approach. Traditionally in bundle-data approach the datapath is accompanied by a matching delay line in the

control path. In our design we made these delay lines adjustable to extend the operational range of the circuit and to make it more power-proportional.

To synthesise a delay line for a particular component first of all we needed to find out its critical path delay under normal operating conditions (a typical corner case in the technology library). This is the required time for a component to have the valid output after computation, hence the acknowledgement signal needs to be generated after it. In our delay line the *delay_init* block (see Figure 4.14) represents this time (minus the delay of all the OR-gates and multiplexers in the line). This block was synthesised using the Synopsys Design Compiler tool. Secondly since the delay of components vary with the supply voltage, we need to adjust our acknowledgement signal, hence the delay line according to models which predict this variability [155]. We also needed to apply 10% margin to these delays for the tool to be able to synthesis a delay and finally equally spread the worst case delay of the unit throughout the whole delay line. We implemented a matching delay line to be adjustable in a wide range of operating voltages (from the nominal down to the threshold voltages).

According to these models, which show the dependency between the supply voltage and the gate latency, we can predict how the critical path of the component changes at lower voltage levels, hence we can adjust our delay line accordingly. As the supply voltage changed we need to change the route of the request signal through the needed delay portion blocks (see Figure 4.14) so that the acknowledgement signal will be generated when the component's output data is valid. This routing is done by applying a particular delay code to multiplexers. The simulation results of one of the delay lines accompanied with a merge controller with four independent request inputs is shown in Figure 5.3.

The *code[7..0]* bus represents the delay code in the line controlling the length of the delay. Signals *req1*, *req2*, *req3* and *req4* are the inputs to the merge controller from the main control logic, as we may access the same datapath unit several times during the execution of the same instruction. Once the request comes to the controller it generates the *req_l* signal, which is sent to the delay line. After the delay the *ack_l* signal comes

back to the controller from the delay line and then the needed acknowledgement is send back to the control block.

Most of the delay registers have 8 bit to hold a delay code for the datapath unit, which indeed showed a lack of granularity of the delay line during the measurement process. The current work is focused to reduced the granularity and also to apply a competition detection techniques to remove the this issue completely.

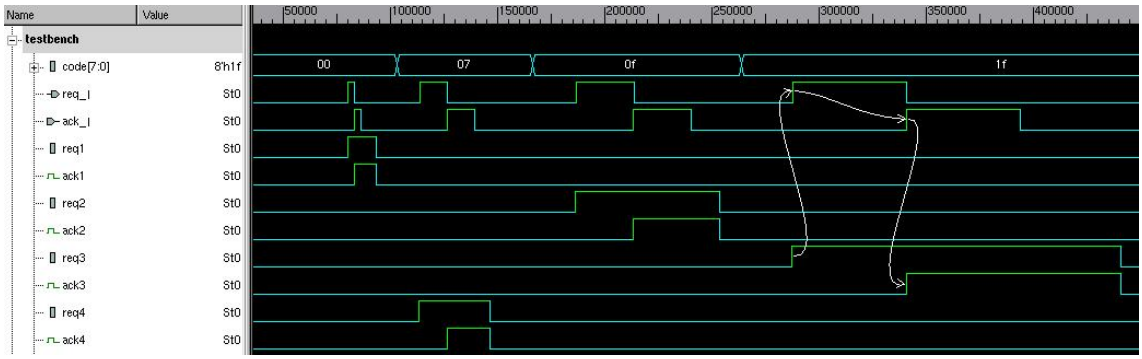
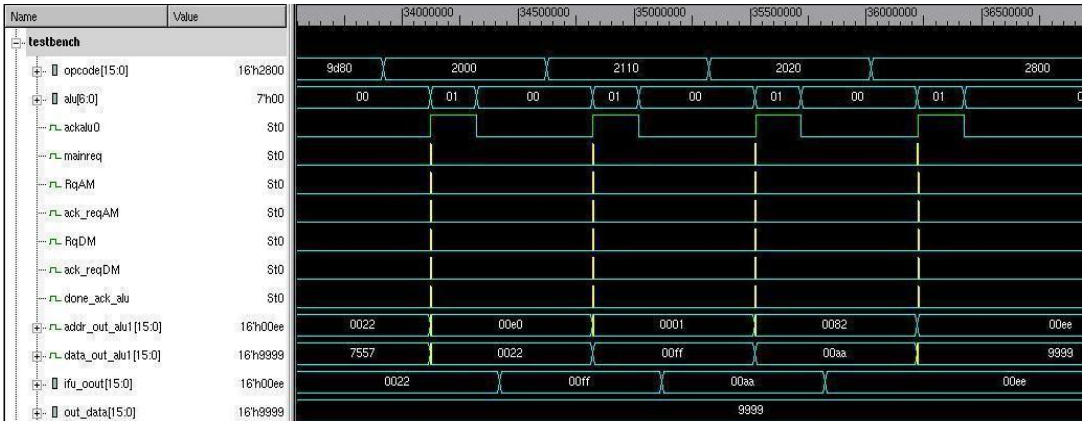


Figure 5.3: Simulation of the merge controller accompanied with an adjustable delay line

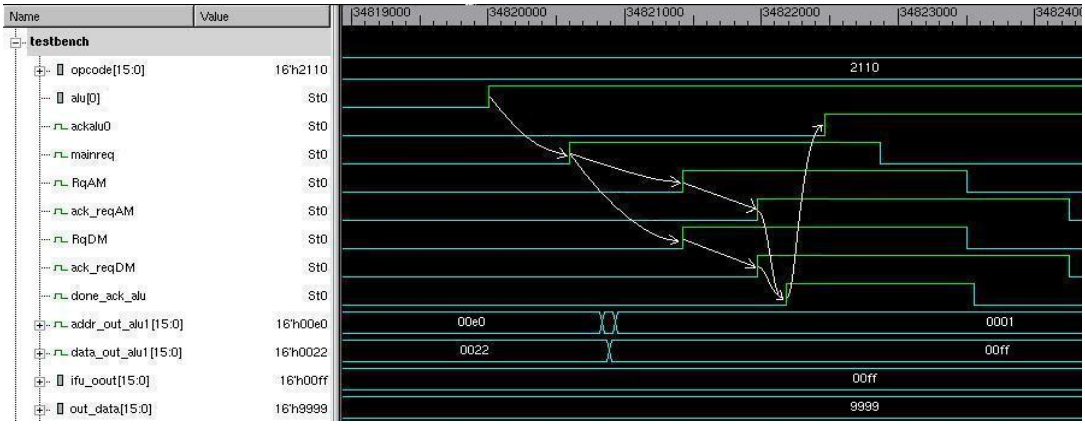
Finally the complete ALU component was synthesised and tested. Simulation waveforms are shown in Figure 5.4. ALU simulation of the instructions from the *class C* (*MOV A, #data*, *MOV Rn, #data*, *MOV DPTR, #data* and *MOV dir, out*) is shown in Figure 5.4(a). Each instruction has its opcode stated in *opcode[15..0]* bus: h2000, h2110, h2020 and h2800 respectively[2]. The bus *alu[6..0]* shows how many times the ALU block is used in the specific instruction. In these particular instructions the ALU block is used only once, therefore *alu[6..0]* can only be 0 or 1 and accordingly we have only one acknowledgement send back *ackalu[0]*. For each instruction we have a different destination where we need to move the data: A, Rn, DPTR or dir, therefore we have different addresses loaded in *addr_out_alu1[15..0]* bus.

Figure 5.4(b) shows a detailed representation of an example instruction (*MOV Rn, #data*) execution: after the ALU control unit receives a request signal (*alu[0]*) from the top-level control it starts the execution (signal *mainreq*) according to its CPOG; following a PO from the ALU control we need to write an address of a particular *Rn* re-

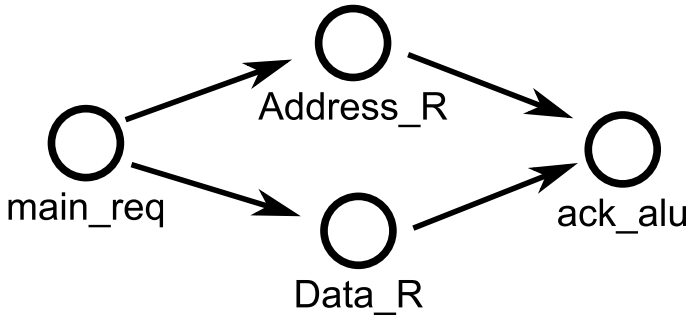
[2]All opcodes and corresponding PO can be found in Appendix A.3.



(a) ALU simulation of instructions from class C



(b) Closer look of ALU simulation of *ADD Rn, #data* instruction



(c) PO representation for the instruction at the ALU-level control

Figure 5.4: Simulation waveforms of the synthesised ALU

gister (in this example it is register 1 (see Table 4.3)) to the Address register (the bus *addr_out_alu1[15..0]*) and data from ROM (this instruction is using immediate addressing mode, so the data is stored in the ROM memory, which is read by the IFU block and holds in *ifu_oout[15..0]* bus) to the Data register (the bus *data_out_alu1[15..0]*) in this example the values are h0001 and h00ff respectively; when acknowledgements from both registers are received the the *done_ack_alu* signal is issued denoting the end of the PO. Finally the ALU control sends an acknowledgement *ackalu0* back to the top-level control.

Another simulation waveform of the instruction *(ADD A, #data* (opcode hCC00)) is shown in Figure 5.4(c). According to the instruction's PO (see Appendix A) there are two requests to the ALU control from the top-level control (*alu[6..0]* bus can be 0 – no requests, 1 – ALU is requested for the first time or 3 – ALU is requested for the second time), so two different acknowledgements *ackalu0* and *ackalu1* are send back when the execution is finished. During the first request according to the PO we need to read the accumulator, hence the ALU only needs to write the accumulator's address (h00e0) to the Address register, so the request *ReqAM* is generated and when acknowledgement (*ack_reqAM*) is received the PO is complete (signals *done_ack_alu* and *ackalu0*). When the ALU is requested for the second time (*alu[6..0]* equals to 3) according to the PO the ALU needs to add two values (buses *a_bl* and *b_bl*), store the result in the Data register and update the PSW register. As this instruction also uses immediate addressing mode, the second value (*b_bl*) h0088 is taken from the *ifu_oout[15..0]* bus, which comes from IFU block (data from the ROM block). Within the ALU block we have a Datapath block, which is in charge of all the logic and arithmetic operations. The *RqAluu* signal shows the start of this block to operate. When all the needed data for the arithmetic operation is ready and it issues the start signal for the adder (*req_bl*). The result of addition is given in the *data_from_bl[15..0]* bus, after we receive the acknowledgement from the adder (*ack_from_bl*) the *ack_Aluu* signal is generated denoting the end of the Datapath block operation. The next stage is to store the result in the Data register (*RqDM*) and update the PSW register (*RqPSW*), when both of these registers issue their acknowledgements, the PO finishes the execution with a signal *done_ack_alu* and finnaly *ackalu1* is send

back to the top-level control.

In the same way the ALU component was verified by executing all the instructions.

5.3.2 PCIU implementation

Section 4.3.2 explained the main features of the PCIU, which is a 16-bit loadable counter. Usually the counter simply increments the IP as the program progresses, however we can also have a branching instruction, when the counter needs to be loaded with a new value.

Figure 5.5 shows the waveform of the PCIU simulation with both loading and counting situations. Signals *count* and *complete_c* represents request and acknowledgement for the counter to count and *load* and *complete_l* for the loading data from the *d[15..0]* bus respectfully. The *q[15..0]* bus is the output of the counter.

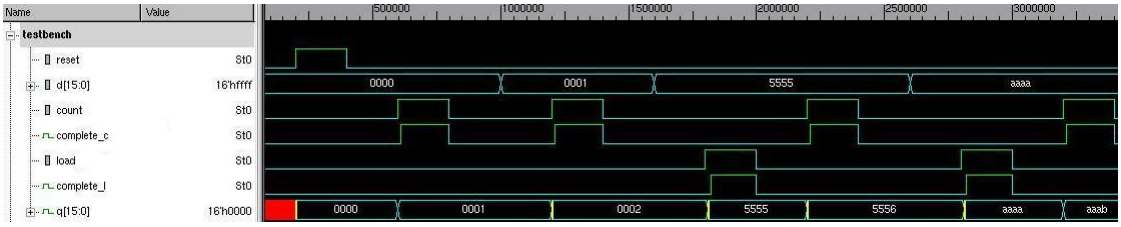


Figure 5.5: Simulation of the PCIU component

5.3.3 Design of the IFU and delay registers

During the execution of the program the main job of the IFU component is to fetch instructions from the ROM block and send them to the processor core for the execution.

Another task of the IFU block is to fetch Delay Codes (as they are placed in the ROM) and store them in the Delay Registers (DR). This procedure is happening during the reset stage of the CPU, as we need to have valid delay codes stored in the DRs before the microprocessor starts executing the main program. Simulation of this process is shown in Figure 5.6.

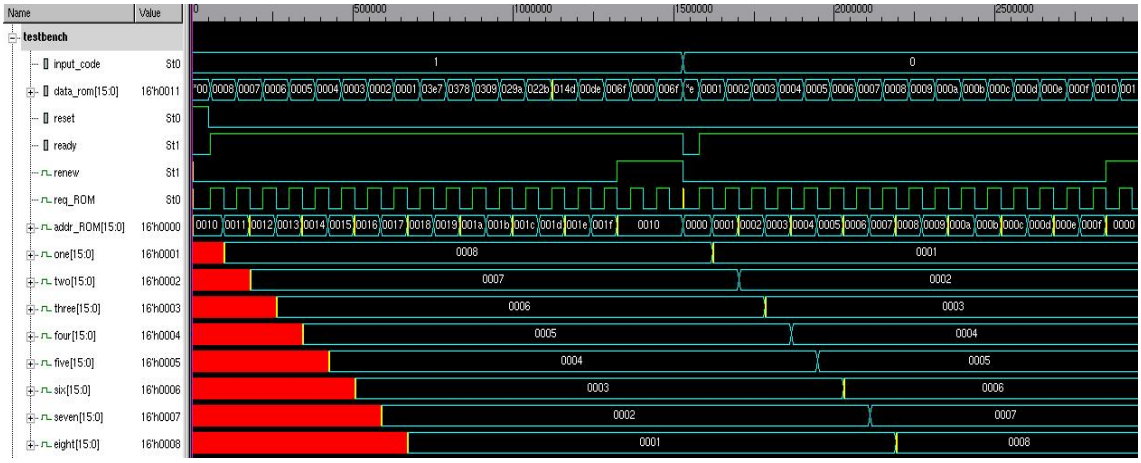


Figure 5.6: Simulation of loading Delay Codes to the Delay Registers

In this simulation the *input_code* bus represents the Delay Bit (Section 4.3.3), which specifies from which set of ROM addresses the Delay Codes will be read (starts either from h0000 or h0010). We introduced these two locations for the Delay code, so that we do not need to reprogram the whole ROM, to change the Delay Code, but we can switch the Delay Bit and fetch a new delay data. After the *ready* signal is received the block starts sending request signals (*req_ROM*) to the memory with a specific address (*addr_ROM[15:0]*); the received data from the memory is shown in *data_rom[15:0]* and then it is written to a needed DR. After the complete set of Delay Codes is read the *renew* signal is issued representing the end of the procedure. The Delay Bit was introduced

This subsection concludes the implementation of the particular components of the microprocessor, now we move to the verification of the complete design.

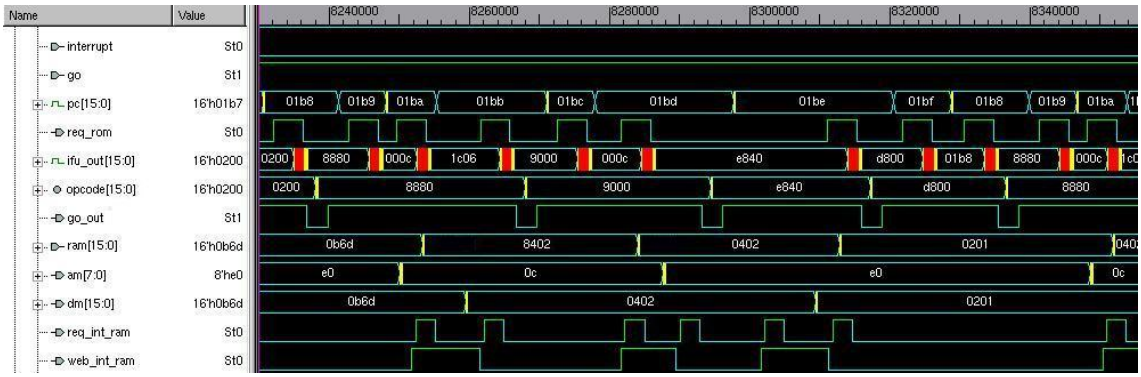
5.4 Verification of the entire chip and sign-off for the ASIC

After we verified all parts of the design separately, the next step before proceeding with layout was the simulation of the complete design, which is addressed in Section 5.4.1. The last step before the design can be sent for the fabrication is place and route (P&R), which afterwards also requires verification.

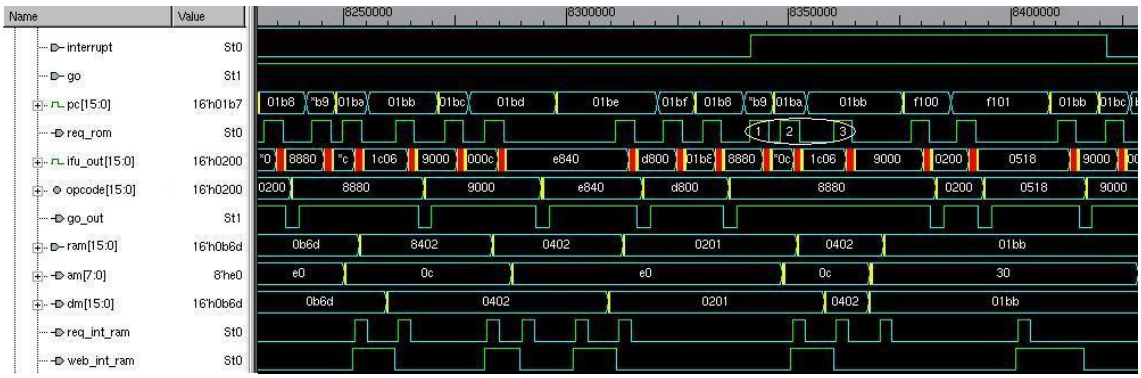
5.4.1 The complete design simulation

Before the design goes to P&R, we need to make sure that all previously synthesised blocks (Figure 4.2) are working correctly together and there are no timing violations. This has been achieved in two ways: by Static Timing Analysis (STA) [22] and Dynamic Timing Analysis (DTA). The STA verifies if the circuits meets its timing constraints, so there are no setup and hold timing violations. However it doesn't check its logical correctness, therefore it is much faster than DTA, which runs an exhaustive gate-level simulation. On the one hand, DTA requires much long time, as we need to apply a high number of input test vectors and check the output ones. On the other hand, it is applicable to both synchronous and asynchronous designs, however STA is more suitable for synchronous approaches.

Figure 5.7(a) shows a waveform with part of the testbench simulation.



(a) Example simulation of the complete design



(b) Simulation with an interrupt occurring

Figure 5.7: Waveforms of the complete design simulation

Table 5.1 shows the code (from a bigger testbench), which simulation is presented in Figure 5.7.

Table 5.1: Additional information for simulations

Program Counter	Opcode	Mnemonic	Machine code
01b8	8880	ANL dir, #data	8880, 000C, 1C06
01bb	9000	MOV A, dir	9000, 000C
01bd	E840	RR A	E840
01be	D800	JMP rel	D800, 01B8
01b8	8880	ANL dir, #data	8880, 000C, 1C06

The *pc[15:0]* bus represents a program counter (PC), which starts in the waveform with the address h01b8. The top-level control sends a request signal (*req_rom*) to the ROM, where first instruction (ANL dir, #data) of this example is located. At that point the previous instruction has finished its execution and the next one starts (signals *go_out* and *opcode[15:0]*). This instruction requires data from the RAM block and the immediate data from the ROM. To read the data from RAM we need the address (h000C) (see the Table 5.1), which is located after the instruction's opcode. When the RAM address is read from the ROM block and written to the Address Register (*am[7:0]*) we can proceed with reading the RAM (the request signal *req_int_ram* and read/write signal *web_int_ram*). The result of the reading is shown in *ram[15:0]* bus (h0402). Concurrently with the RAM reading we also fetch data from the ROM block (the value h1C06). When all the data is ready we can proceed with the AND operation, its result is shown in *dm[15:0]* bus ready to be written to the RAM. Concurrently with the writing process to the RAM we are ready to fetch the opcode of the next instruction (h9000). After this the execution of the instruction finishes (signals *go_out*).

In the same way the other 3 instructions are being executed. Notice that the JMP instruction is updating the PC with a new value (h01B8), i.e. this loops the execution of the whole testbench.

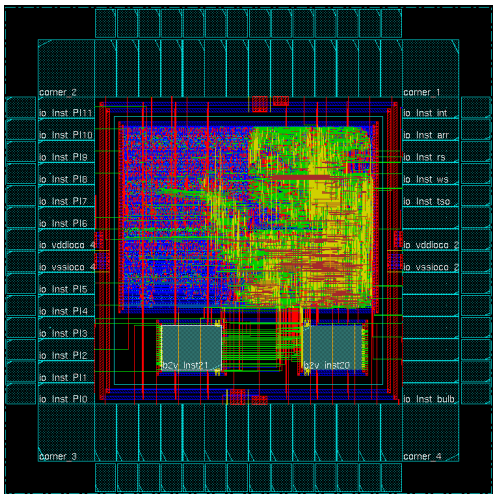
The next example (Figure 5.7(b)) shows the same testbench, but with an interrupt happening (signal *interrupt*). In Section 4.3.7 we described the main details of the in-

terrupt handler, where it was mentioned that the handler's PO starts at the end of any instruction's PO, if an interrupt had occurred. This can be seen in the waveform when the interrupt rises during the execution of *ANL dir, #data* instruction (opcode h8880) and at the end of the execution (the third request to the ROM, see Figure 5.7(b)). Similar to the previous example (without an interrupt), the request fetches the next instruction (opcode h9000) of the program, however as we have an interrupt, first the PC is saved in the Stack and then the entrance address of the interrupt handler is loaded to the PC. When we start to execute the handler, the interrupt signal can still be high, however this doesn't affect the execution. In this example our handler contains just one instruction (opcode h0200). At the end of the handler we have a *RETI* instruction (opcode h0518) to initiate the exit from the interrupt. At this point we read our old PC from the Stack and continue the execution of the main program (opcode h9000). Note that the programmer needs to take care of all the important resisters (A, B, DPTR, etc.), which can be overwritten during the interrupt handling, e.g. by storing the context in the Stack.

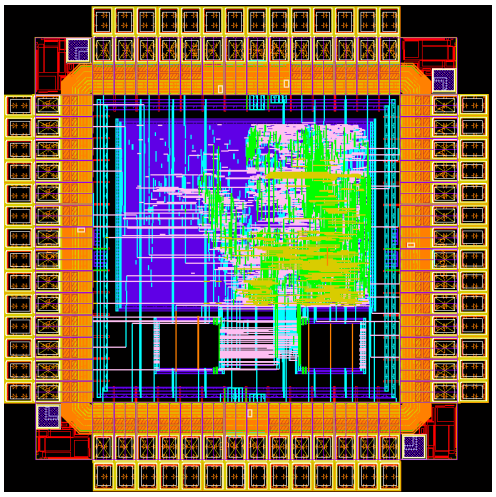
5.4.2 Chip layout and final verification

After the synthesised design has been fully simulated and verified we proceeded with its layout and post-layout verification.

Before handing our synthesised netlist to the layout tool we needed to specify the chip's floorplan, power domain regions and I/O pins. As we have several separate netlists to layout (the main design (the Control logic and the Datapath) and two RAM blocks) it is important to define a region for each power domain and their location in the chip. Following the floorplanning we imported out netlists into the Encounter tool, which has the physical information about cells from the specified technology library and a predefined floorplan where these instances will be places. During the P&R the tool optimally places each instance in the chip's floorplan and routes internal connections within each instance as well as between them and external ones to the outside I/O pins. The final view of the design after P&R is shown in Figure 5.8(a).



(a) The abstract view of the microprocessor after P&R



(b) Design view with a corresponding cell layout

Figure 5.8: Various views of the design during P&R

After our design was successfully placed on the chip, we needed to verify that its functionality was not affected during the P&R. Tools for the gate-level simulation usually do not take the disposition of the blocks in the design into account, which can significantly effect the timings and therefore the correctness of the entire design. After the P&R procedure the position of the components is fixed and we can extract the information about wire delays and proceed with post-layout simulation. The Encounter tool allows us to go through the Parasitic Capacitance Extraction and export a Standard Delay Format (SDF) file, which provides information about wire delays for the simulation and timing analysis tools. After that we rerun all the simulation testbenches and verify that our CPU gives the correct results.

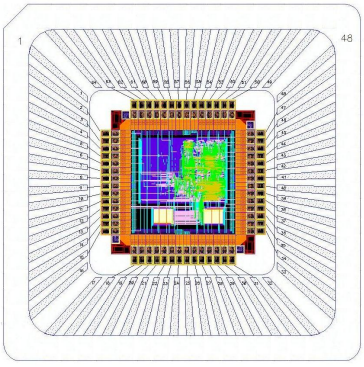
At the next stage it is essential to export our design from the Encounter to a special file in Design Exchange Format (DEF), which is the input format for the Virtuoso tool. This tool replaces an abstract view of each cell (from the Encounter) with its actual layout. Figure 5.8(b) shows the design view after it was substituted with the actual cell layout.

Every manufacturing company has their own restrictions on ASIC fabrication. Therefore before sending design to them we needed to satisfy the restrictions and make sure

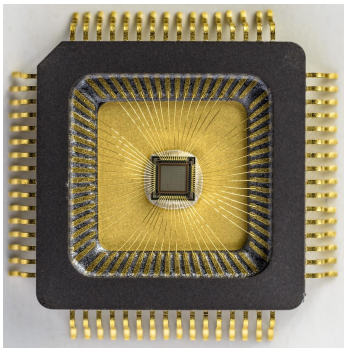
that our chip can actually be fabricated. To proceed with this we needed to export our design from the Virtouso to a special Geometric Data Stream (GDSII) file, which can be read by the verification tool. We used the Calibre tool by Mentor Graphics to go through DRC and LVS checks. There were several iterations between us and the manufacturer before they actually accepted our design.

The microprocessor was fabricated in the *130-nm* CMOS process using the standard cell library from *STMicroelectronics [140]* semiconductor foundry. This technology library was provided by *Circuits Multi-Projects* (CMP) [39] service company. Collectively, the main design block and two RAM components occupy 2.95 mm^2 of silicon.

Figure 5.9(a) presents the bonding diagram of the chip (more details about each of the pin on bonding diagram is shown in Appendix D) and Figure 5.9(b) shows a photo of one of the 25 prototypes received.



(a) The bonding diagram of the chip



(b) Photo of the fabricated and packaged chip

Figure 5.9: Bonding diagram and packaged ASIC

5.5 Testing board

Before the arrival of the chip we started to prepare a Printed Circuit Board (PCB), for testing our microprocessor. Several important aspects needed to be considered during its development:

- an off-chip EPROM, which holds the main program and delay codes.
- there are “test_mode” and “work_mode” (see Section 5.3), in which our processor

can operate, so the PCB needs to provide the ability to control the "mode_select" and "ram_select" pins and connect specific signals to particular pins.

- convenient access to the rest of pins, i.e "reset", "go", "external data", "delay mode", "calculation mode", "interrupt", etc. (see Figure 5.9(a)).

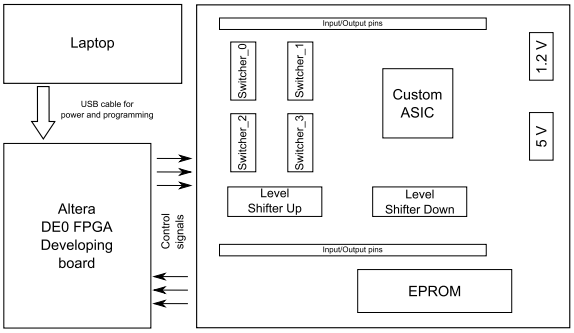
In the light of the above we decided to connect our PCB to an FPGA (Altera DE0 developing board [12]), which would provide a more convenient control over these specific pins and modes; moreover the FPGA board can also be used as a ROM. However the use of FPGA has a drawback, as it requires a 5V power supply, but the nominal voltage of our design is 1.2V. Hence we needed to provide voltage level shifters between the FPGA and the fabricated ASIC. At this stage we started to search for all the required components for the PCB and to order them from the IC's suppliers like Premier Farnell [123], Radio Spares (RS) [128] as well as from the University's internal stock. Figure 5.10(a) depicts a simplified diagram of the PCB. Figures 5.10(b, c) show a pictures of the fabricated PCB with all the components on it and the FPGA board connected to it.

After checking the functionality of the PCB, we moved to the stage of testing, recording of the measurements and their analysis.

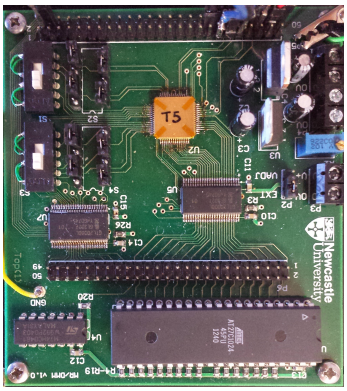
5.6 Measurements and results

An FPGA development board is used to control our PCB with the microprocessor. So before measurements we needed to develop a design for the FPGA board according to the aspects discussed in the previous section and test its functionality.

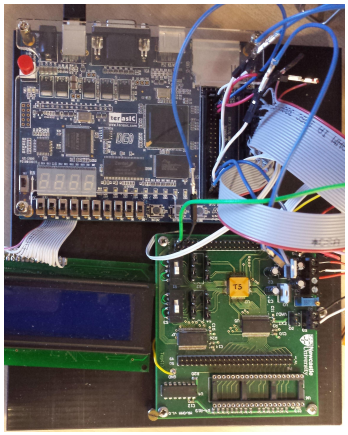
At his point we can connect both of the power supplies (1.2V and 5V) to the PCB, program the FPGA and run the 8051 CPU. However before the main CPU program can be executed we need to load the Delay codes. So during the Reset stage (signal "reset" in Figure 5.11), the "Delay Bit" was set to "0", the CPU started sending requests ("req") and addresses ("address") to the ROM (through pins "output[15:0]" and "ROM_REQ" see Figure 5.9(a)). The "data" bus and "ack" signal are coming from the FPGA board to the chip as data from the ROM and the acknowledgement signal. Figure 5.11 shows this



(a) Schematic diagram of the PCB



(b) Photo of the PCB



(c) Picture of the FPGA board connected to the PCB

Figure 5.10: The PCB and FPGA boards

process captured by a digital signal analyser connected to the PCB.

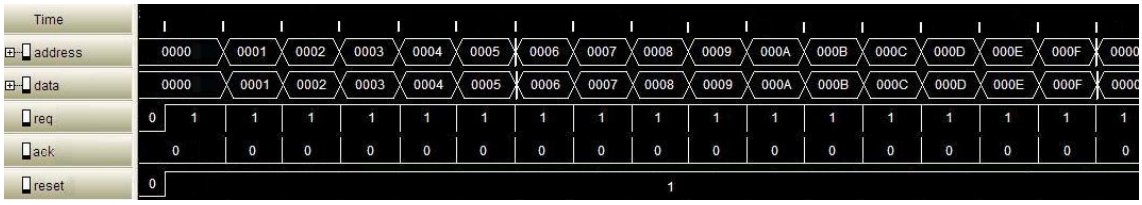


Figure 5.11: Loading of Delay registers captured by Digital signal analyser

The Delay codes can also be loaded by using the Scan chain. First we need to switch the chip into a test mode (by enabling the "mode_select" pin), then load a test vector of Delay Codes using the Scan_in and Scan_clk inputs (see the floorplan of the chip (Appendix D) and pin reassignment for the modes (Appendix E)). Using the Scan chain we can also check the correctness of the previously loaded Delay codes by enabling the

"mode_select" pin and checking the Scan_out pin every Scan_clk impulse. Once all the codes in the register are loaded we can run the processor.

First we fill the ROM with NOP instructions to see if the execution follows its PO (see Appendix A.21). From outside the chip we can observe the request to ROM block and the "GO_OUT" pin, which represents the end instruction's execution (see Figure 5.7). Measurements were taken on several power supply voltages: 1.28 V, 1 V, 0.7 V, 0.5V and 0.25 V. Oscilloscope screenshots are shown in Figure 5.12.

For the execution of NOP instructions we need to increment a PC and then fetch the next instruction from the ROM. As the variable voltage will affect the execution time of the PC and ROM, for each of the voltage levels we needed to change the delay codes. During this experiment we noticed that the PC (the address which is coming from the chip) starts failing when the voltage goes below 0.7V – it gets stuck at fetching the same address forever. However, the rest of the control logic synthesised using the CPOG model continues to operate correctly down to 0.25V.

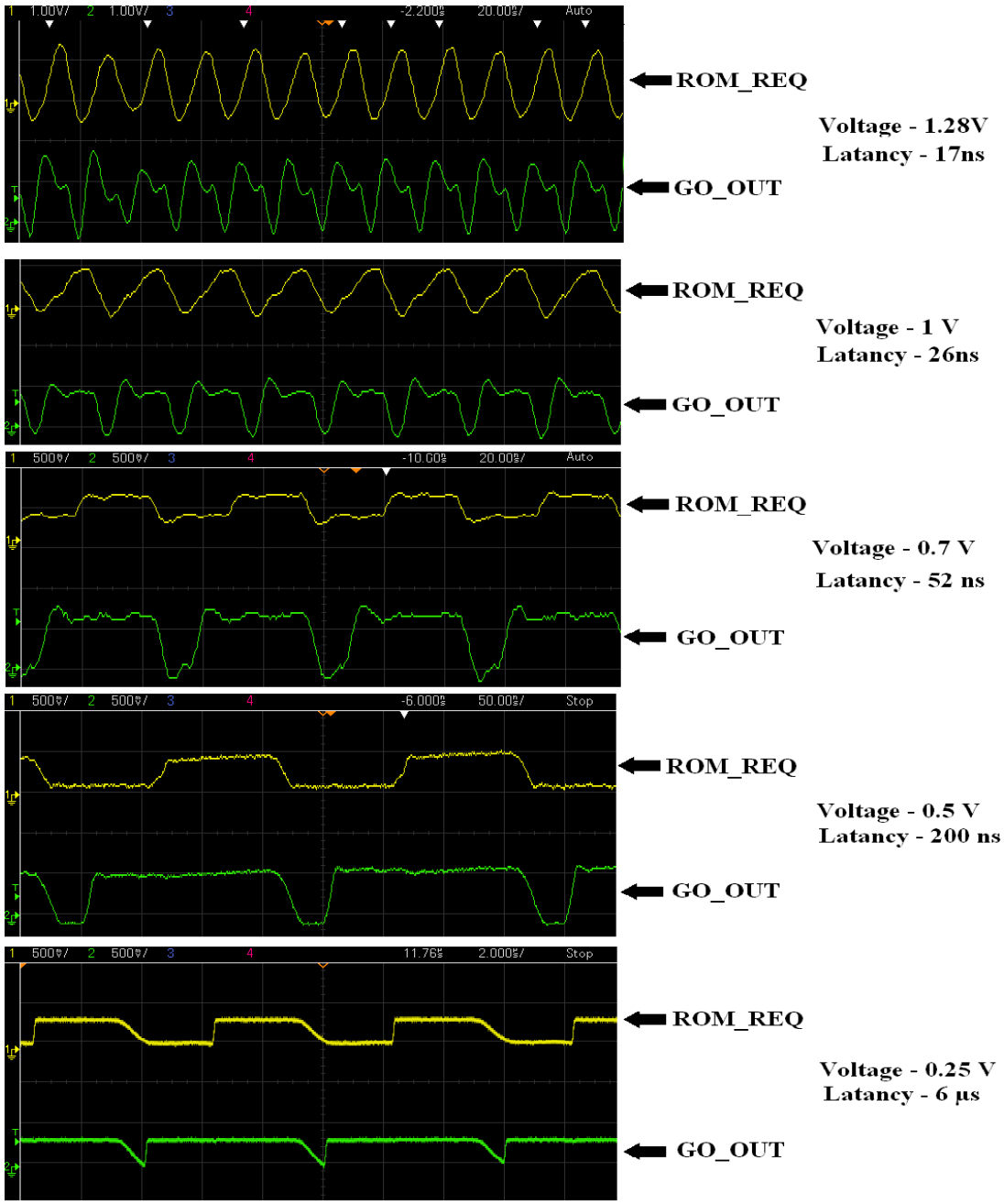


Figure 5.12: Oscilloscope screenshots for NOP instructions on variable voltage

In the same way we tested all the instructions classes. Figure 5.13 shows oscilloscope screenshots for a loop execution of example instructions from different PO classes (see Appendix A). Depending on instruction's PO we can have different numbers and sequences of "ROM_REQ" coming from the chip.

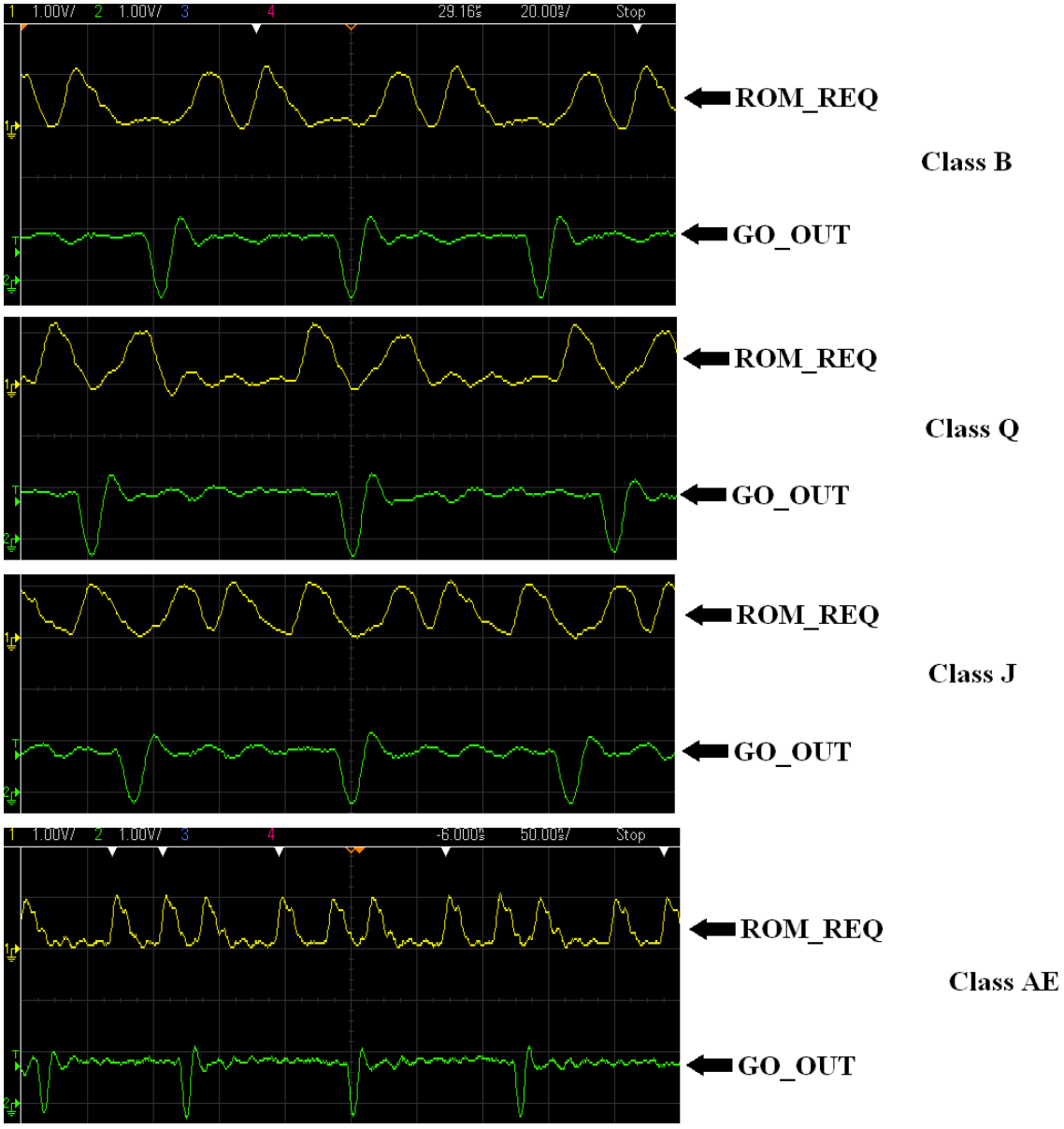


Figure 5.13: Oscilloscope screenshots of different instruction's execution

After testing all the instruction the next step was to characterise the processor performance and power consumption depending on particular voltage levels. Moreover, having two implementations of each computation unit (adder, multiplier and divider) we explored the ability to switch between high performance and low power datapaths and compare the performance and power consumption. Following plots representing measurements from example instruction execution:

- "NOP"

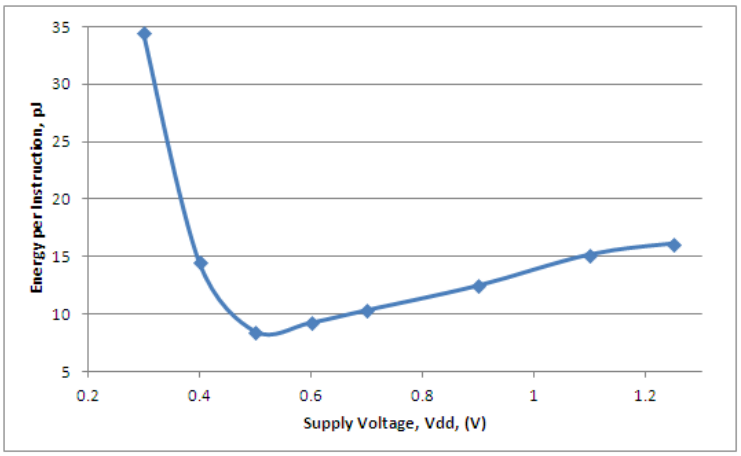


Figure 5.14: Measured EPI when Vdd changes for NOP instruction

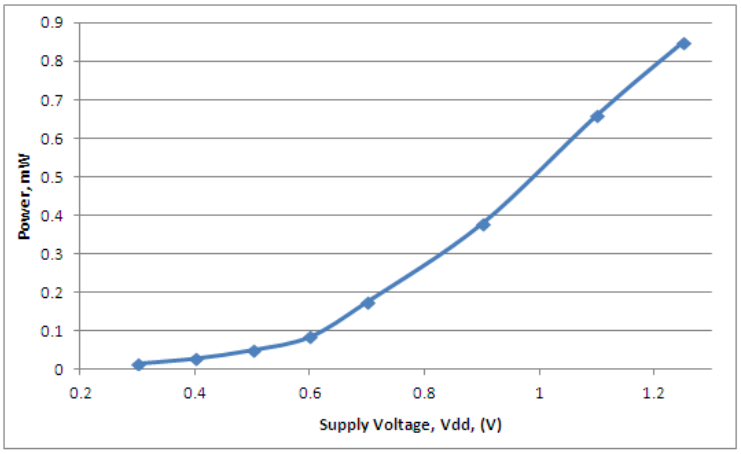


Figure 5.15: Measured power consumption when Vdd changes for NOP instruction

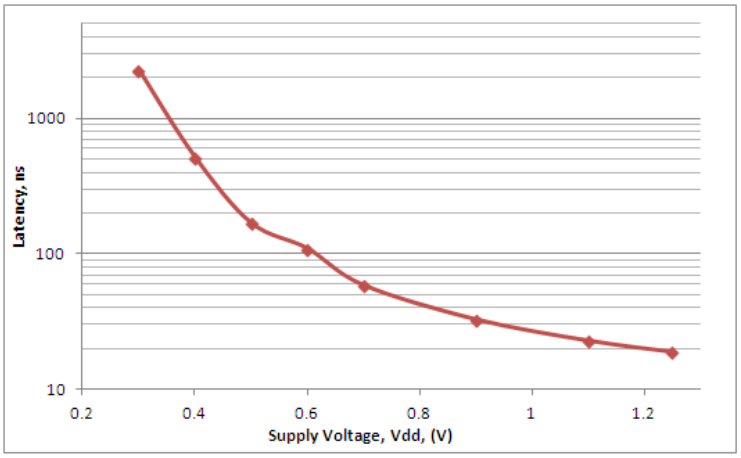


Figure 5.16: Measured latency when Vdd changes for NOP instruction

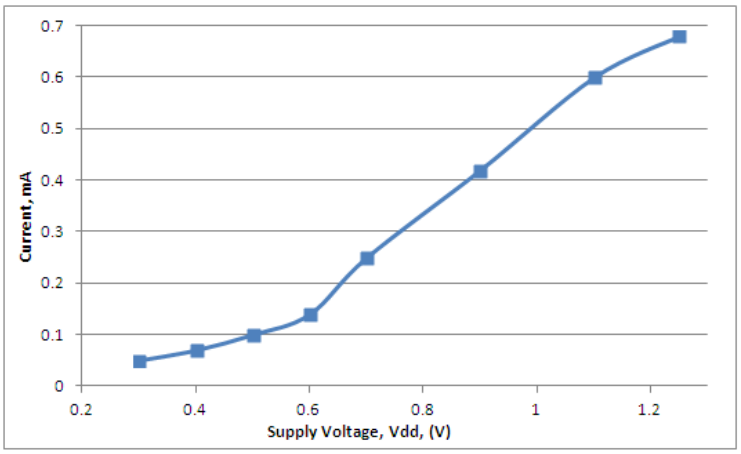


Figure 5.17: Measured current when Vdd changes for NOP instruction

- **"SJMP rel"**

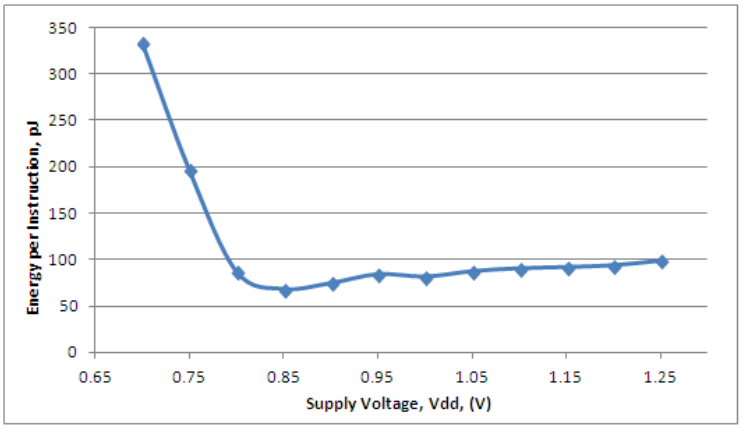


Figure 5.18: Measured EPI when Vdd changes for SJMP instruction

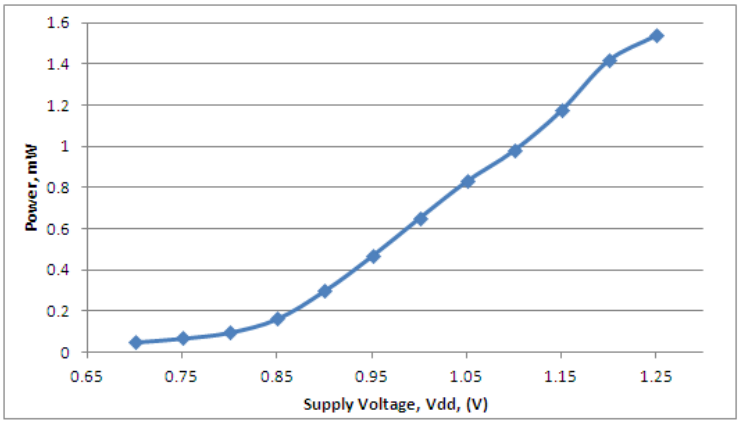


Figure 5.19: Measured power consumption when Vdd changes for SJMP instruction

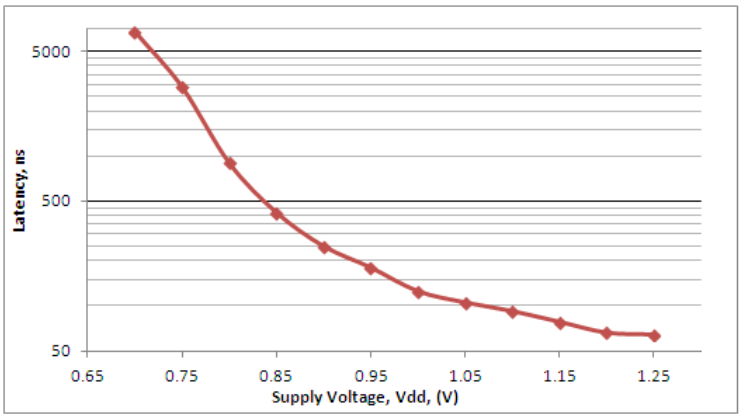


Figure 5.20: Measured latency when Vdd changes for SJMP instruction

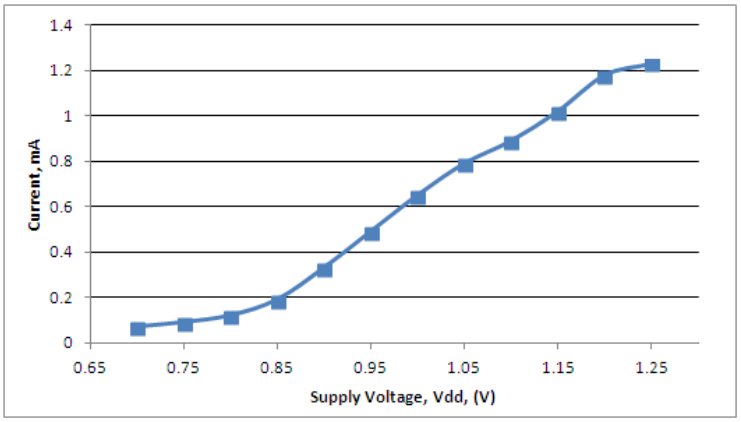


Figure 5.21: Measured current when Vdd changes for SJMP instruction

- **"ADD A, #data"**

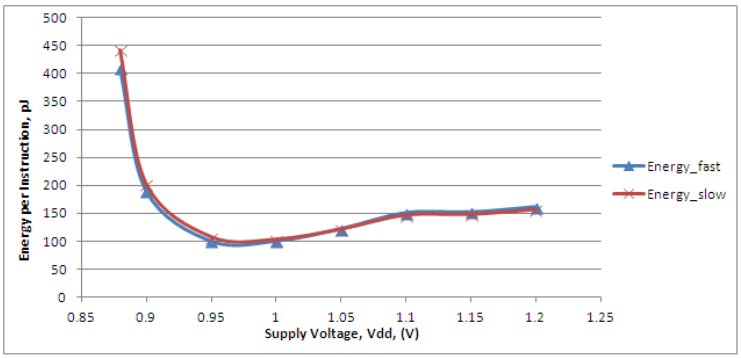


Figure 5.22: Measured EPI when Vdd changes for ADD instruction

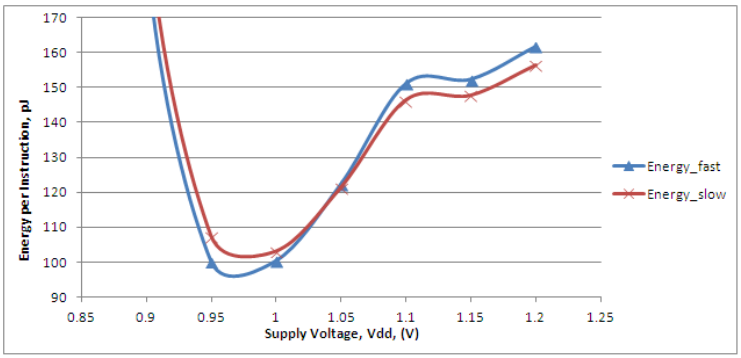


Figure 5.23: Closer look of a measured EPI when Vdd changes for ADD instruction

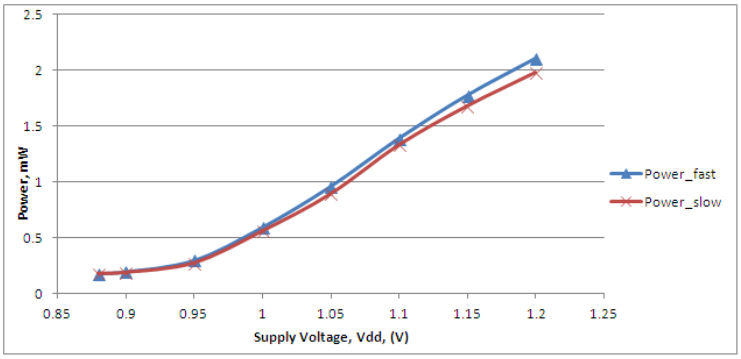


Figure 5.24: Measured power consumption when Vdd changes for ADD instruction

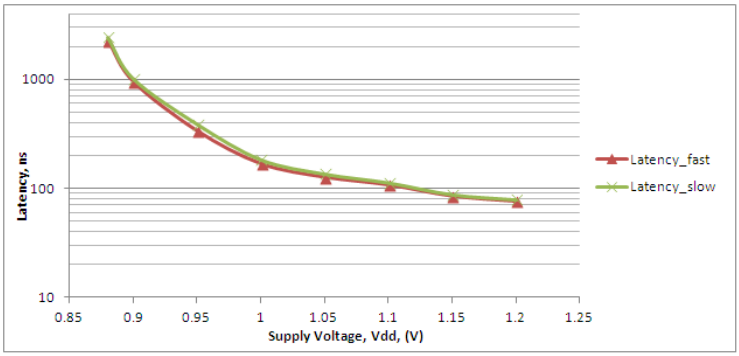


Figure 5.25: Measured latency when Vdd changes for ADD instruction

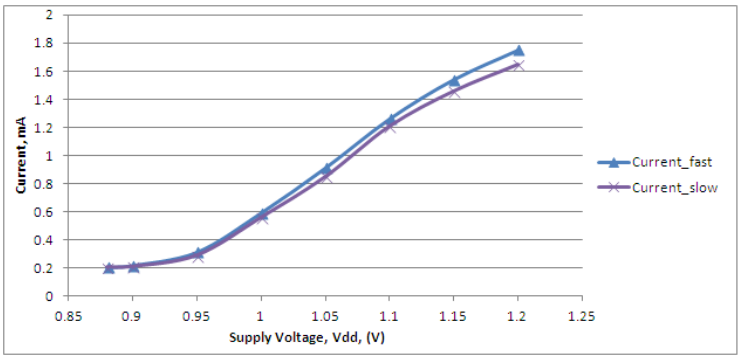


Figure 5.26: Measured current when Vdd changes for ADD instruction

- "MUL A, B"

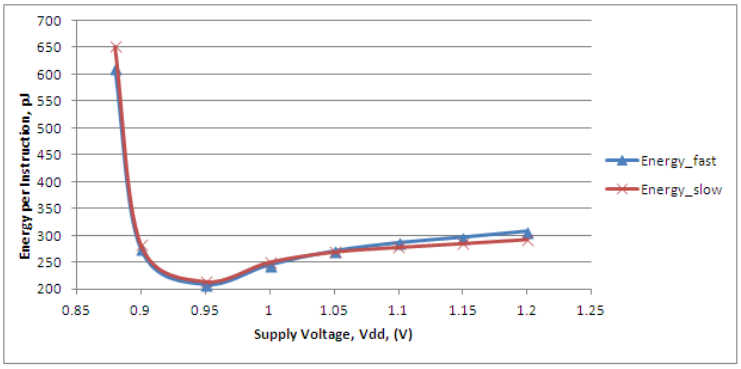


Figure 5.27: Measured EPI when Vdd changes for MUL instruction

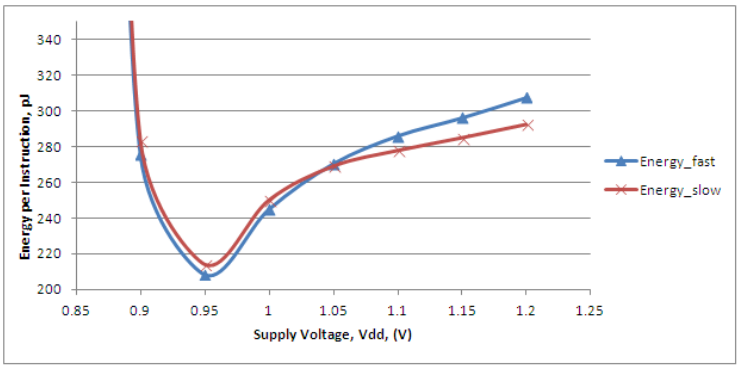


Figure 5.28: Closer look of a measured EPI when Vdd changes for MUL instruction

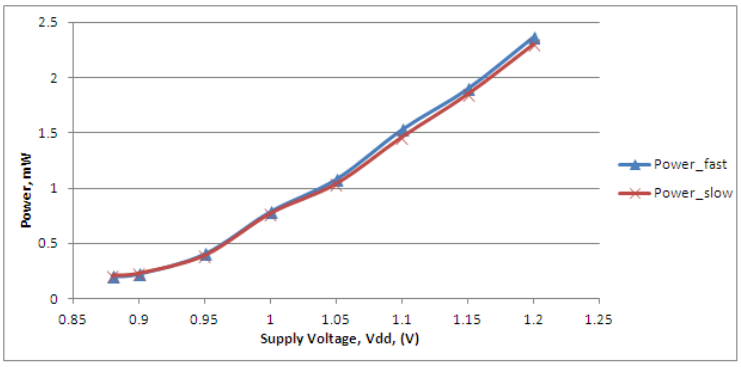


Figure 5.29: Measured power consumption when Vdd changes for MUL instruction

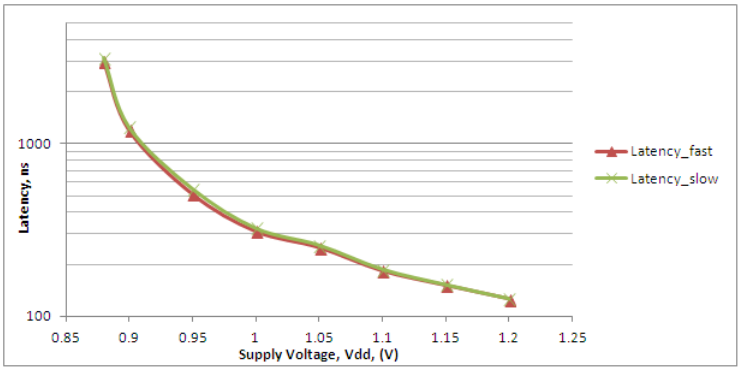


Figure 5.30: Measured latency when Vdd changes for MUL instruction

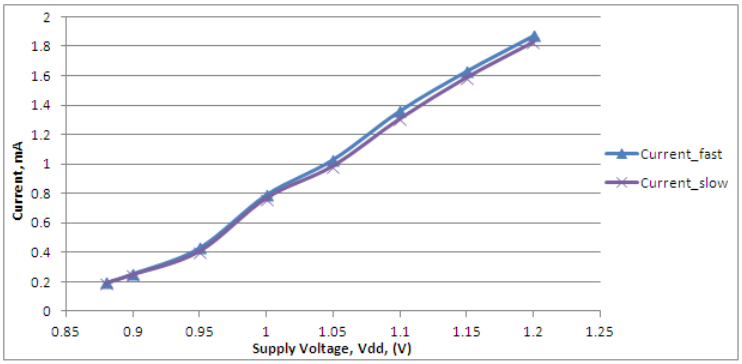


Figure 5.31: Measured current when Vdd changes for MUL instruction

The execution of different instructions requires different computational units to operate. Since some of the units have different ranges of operating voltages, e.g. on-chip RAM components can not work below 0.87 V or the PC operating voltage is above 0.7 V, different instructions have also a different operating voltage range (i.e NOP – from 0.25V up to 1.25V; ADD – from 0.87V up to 1.25V).

Analysing the presented plots we can see that different instruction have different minimum energy point, e.g. "SJMP rel" instruction at 0.5 V (Figure 5.14) and "MUL A,B" instruction at 0.95 V (Figure 5.28). Therefore depending on how often the programmer will use a particular instruction at a specific voltage level the total power consumption may vary.

Interesting results were obtained during switching between two sets of computational units. The difference in power between the fast and slow modes for a particular voltage level may be small, however depending on how often the programmer is using a particular mode this difference accumulates and eventually results in significantly different total power consumption. Surprisingly we also noticed that low-power computational units do not always consume less energy (Figure 5.23, 5.28). At a particular voltage level the energy consumption of the instruction increases. This is due to the leakage in the datapath units, which are used for the instruction. In this aspect the use of low-power components is not always optimal in a wide range of supply voltages. It is reasonable to use them on higher voltages, when a dynamic power consumption is dominating, however once the computational latency increases, hence the leakage, then it is more efficient to continue computation in high performance mode.

One can notice that there is not much difference between high performance and low power units. There are several reason why that is happening: i) represented results are measured of the whole chip, not the particular computational units. That means that other components, e.g. power-hungry blocks like memory, can have a greater effect on the total numbers and "mask" the difference of these modes. ii) originally the difference of these computational modes are caused my different algorithms used in arithmetic units, however it might be the case that the chosen algorithms are not the best, so we can't see much difference in the measurements. Our current work is focused on further research of the algorithms to find a better option.

We continue the measurements and target to get results for more representative test-benches like Dhrystone V2.1 [124]. Simulation results of this testbench and comparisons with other implementation of Intel 8051 are summarised in Table 5.2.

Table 5.2: Performance comparison with other 8051 versions

Processor	Technology	MIPS	Average power, mW	MIPS per W	Energy, pJ per instruction
Sync_80C51 [151]	3.3V, 350nm	4	40	100	10000
Async_80C51 [151]	3.3V, 350nm	4	9	44	2250
H8051 [42]	3.3V, 350nm	4	44.7	89.5	11175
Lutonium [96]	1.8V, 180nm	200	100	1800	500
DS89C420[125]	1.1V, 350nm	11	18.52	600	1684
Nanyang_A8051 [35]	1.1V, 350nm	0.6	0.07	8000	130
Lutonium [96]	1.1V, 180nm	100	20.7	4830	207
Proposed 8051	**1.2V, 130nm**	**1.5**	**0.74**	**2027**	**493**

The table compares the following implementations of Intel 8051 microprocessor. Sync 80C51, H8051 and DS89C420 are synchronous designs - the first two employ non-pipeline architecture and the last one has pipelined architecture. Async_80C51 is an asynchronous counterpart of Sync_80C51 design. Lutonium is another asynchronous implementation which utilises a highly parallel processing with a deep pipeline architecture - therefore its high MIPS rate and power consumption. On the contrary, Nanyang_A8051 is a self-timed, ultra low power (and thereby low performance) implementation. Despite the technology difference, which can be scaled to the same denominator, our implementation is clearly placed between ultra low power design (runs twice as fast) and the highly parallel version (consumes 20 times less power) [130].

5.7 Summary

In this chapter we explained the main details of the implementation, verification, fabrication and testing our Asynchronous Intel 8051 microprocessor. The implementation of

each component of the processor was divided into several stages (Section 5.1), which is vitally important in developing, verifying and eventually fabricating a correctly functioning design.

The implementation of the ASIC was divided into two main parts: the control part (Section 5.2) and the datapath (Section 5.3) design, which then follows with the complete chip verification (Section 5.4).

After the die has arrived it was important to design the environment where we can proceed with its testing (Section 5.5). During the verification stage we checked that all the instructions are executed correctly, which demonstrates the correctness of the proposed microprocessor's design flow (Chapter 4).

Analysing the above results we can conclude that depending on application requirements and/or the power budget a user can adapt our microprocessor core for a particular purpose. Extending a traditional datapath structure to work in several operating modes along with its asynchronous nature enables our microprocessor to work in a wide range of operating voltage (from 1.25V down to 0.25V) and environmental conditions.

One might want to compare the presented measurements in terms of trends addresses on Figure 1.2, which shows two power-proportional designs each of which was developed to work in a different power domains. In this work we discus the developing process to implement a system that can adapt to different power levels and work in a wide range of supply voltages. If we build this plot for our system and apply different computational modes, we would see similar trends, so at some degree it was achieved. However due to several addressed issues, e.g. not clear difference between computational units, it would be more closer to a straight line rather than to "gull wing" shape, as on the figure.

The next chapter summarises the contribution of the thesis and discusses further research directions.

Chapter 6

Conclusion

This thesis presented a design flow for the development of microprocessor instruction set architectures, which can be altered to suit a particular hardware platform or a particular operating mode. The feasibility of the methodology, novel design flow and a number of optimisation techniques were proven in a full size asynchronous Intel 8051 microprocessor and its demonstrator silicon. Our implementation shows a competitive result and the ability to adapt to a wide range of operating voltage and environmental conditions.

This chapter summarises the key contributions of this thesis and outlines areas of future research.

6.1 Main contributions

It was demonstrated in Chapters 1 and 2 that the current ICs and particularly microprocessor design flow shifts emphasis from high performance towards energy-efficient and power-proportional solutions, as energy and power turn from optimisation criteria to a guiding principle. In this work, several methods and techniques are proposed to address the problem of designing a power-proportional microprocessor capable of on-line adaptation to varying operating conditions and application requirements.

The essential contributions in this thesis are the following:

- **A design flow for the development of ISA for a microprocessor (Chapter 3).**

The key difficulties in designing ISA is the necessity to comprehend and deal with a large number of instructions, whose microcontrol implementation may be altered to suit a particular hardware platform or a particular operating mode. We demonstrate that a novel CPOG formalism (Section 2.2) is a versatile technique enabling efficient specification of a processor ISA. Crucially, this formalism is a convenient tool for carrying out transformations to the ISA, as these transformations operate on a CPOG specification rather than on the instruction set itself and thus their complexity does not depend on the number of different instructions.

On the basis of a simple example (Section 3.5), we demonstrate how the application of this formalism can be expanded to capture different hardware configurations and operation modes. Further, we prove the correctness of CPOG constructs (Section 3.5.2) with respect to the given functional ISA descriptions using the Event-B model (Section 3.4.1).

- **Development of an adaptive and reconfigurable system with run-time adaptability on the base of an asynchronous Intel 8051 microprocessor (Chapter 4).**

  To demonstrate the feasibility of the introduced design flow on a well-known and sophisticated example and to introduce a new power-proportional criterion in the system design flow we implement a novel asynchronous of 8051 microprocessor.

  Adaptation of the CPOG methodology for both microcontroller blocks (the Top level and the ALU control) significantly simplifies and accelerates their development and validation process. Transformation to the original ISA, e.g. expansion or contraction, is done on the CPOG specification, which simplifies the whole ISA design flow.

  We introduce some new features to our design: i) an extended datapath (Section 4.4.1) with pairs of computational units, each being optimised to work in a specific regime, one optimised for energy consumption and the other one for performance; ii) adjustable matching delay lines (for the bundled data protocol in datapath), which provide a robust operation of the circuit in a wide range of supply voltages (Section 4.4.2); iii) we exploit the fact that the datapath was extended for

a multi-modal operation to provide fault tolerance features (Section 4.4.3); etc.

- **Testing of the design by implementing a proof-of-concept ASIC and evaluating its performance and power consumption (Chapter 5).**

  We implement the proposed asynchronous Intel 8051 microprocessor as a proof-of-concept ASIC. The chip went through a series of tests and evaluation stages including behaviour simulation and an FPGA-based validation. A dedicated PCB board was fabricated along with the chip to provide a convenient testing environment. The control over ASIC was delegated to an external FPGA development board.

  Experimental results proved the feasibility of the proposed design flow to construct a full-size ISA of a commercial microprocessor. It was also shown that by applying a specific operating mode to the extended datapath we can adjust our microprocessor core for a particular application requirements and power budget. Moreover this extension provides not only an adaptable, but also a fault tolerant operation.

The process of developing this design flow and implementing our reconfigurable microprocessor core unveiled several other goals of future research discussed in the next section.

6.2 Future research directions

There are three main directions where this research can go further: i) extension of the CPOG formalism to enable automated scheduling of a particular instruction; ii) several improvements of our microprocessor implementation; iii) investigation of a new realisation of logic gates in the domain of near-threshold voltages.

- In Section 3.5 on a simple example we demonstrate how CPOGs can be used for capturing different hardware configurations and operation modes in the execution of a single instruction. Following this idea we can push the boundaries of power-proportionality even further in automating scheduling of instruction execution in

terms of power/latency trade-off: based on the current operation mode and characteristics and availability of datapath components, the microprocessor can schedule the units in the appropriate partial order.

- Several further improvements can be made in the current CPU implementation:
  - Design of the microprocessor core using a different asynchronous circuit class, such as QDI approach, which provides a built-in completion detection features, as opposed to the bundled data approach.
  - Investigation and implementation of other techniques to reduce power consumption, such as power gating.
  - Research in energy-efficient usage of memory. The current RAM block is unable to work below 0.87V, therefore we can either substitute it with a self-timed SRAM[1], which is capable of operating at lower voltages or go even further by applying a newly developed adaptive technique, which allows the CPU to be switched into ultra low power (and low functionality) mode. The processor will be restricted to use only a small number of specific internal registers rather than the full capability of its memory bank.
  - Interface our chip to the sources of harvested power as well as to the self-powered sensors which can raise interrupts.
- It is a well-know fact that usually more energy is consumed on the moving data around rather than by the computation itself. Although in this work we didn't measure these two aspects, but it would be really interesting to measure and compare these two energies. In a very near future work we are planning to make this comparison.
- Finally, the reliability of the main control and datapath structures can also be improved by utilizing high-reliability logic with low fan-in gates. This may have overheads in terms of area, however it significantly improves the circuit reliability in the domain of near-threshold voltages.

[1] Currently we have a fully working SRAM chip developed in our research group [20]

Appendix A

PO representation of the 8051 instruction Set

This appendix outlines all the instructions from the original 8051 ISA in partial order representation. All 255 original plus two additional instructions are grouped into 37 different classes, which are enumerated in an alphabetic order. Each of the following Section explains a particular class of instructions, shows its PO in the Top-level and ALU microcontroller and outlines all the instructions in this particular class.

A.1 Class A

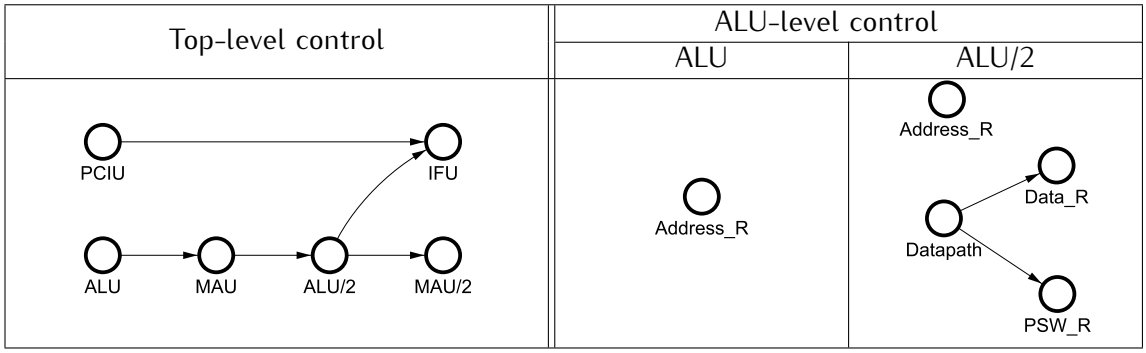


Figure A.1: PO representation for instructions from class A

Table A.1: List of all instructions from class A

Mnemonic	Opcode		Function
	binary	hexadecimal	
MOV A, Rn	1111100100010000	F910	(A) := (Rn)
MOV Rn, A	1111100100000010	F902	(Rn) := (A)
SWAP A	1111000000000000	F000	exchange of accumulator's tetrads
DA A	1110100010000000	E880	correction of the accumulator
INC A	1110000000000000	E000	(A) := (A) + 1
DEC A	1110000001000000	E040	(A) := (A) – 1
INC Rn	1110000100010010	E112	(Rn) := (Rn) + 1
DEC Rn	1110000101010010	E152	(Rn) := (Rn) – 1
INC DPTR	1110000000100100	E024	(DPTR) := (DPTR) + 1
CPL A	1111000001000000	F040	inversion of the accumulator
RL A	1110100000000000	E800	rotation of the accumulator one bit to the left
RLC A	1110000010000000	E080	rotation of the accumulator one bit to the left through the carry flag
RR A	1110100001000000	E840	rotation of the accumulator one bit to the right
RRC A	1110000011000000	E0C0	rotation of the accumulator one bit to the right through the carry flag

A.2 Class B

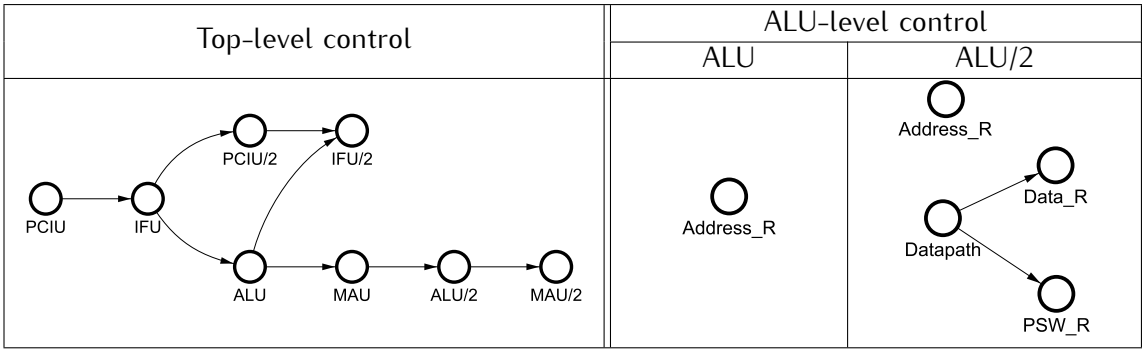


Figure A.2: PO representation for instructions from class B

Table A.2: List of all instructions from class B

Mnemonic	Opcode		Function
	binary	hexadecimal	
MOV A, dir	1001000000000000	9000	(A) := (direct)
MOV Rn, dir	1001000100000010	9102	(Rn) := (direct)
INC dir	1001100000000000	9800	(direct) := (direct) + 1
DEC dir	1001100010000000	9880	(direct) := (direct) – 1
CPL bit	1001110000000000	9C00	bit inversion
ANL C, bit	1001111000000000	9E00	C:=C&bit
ANL C, /bit	1001111010000000	9E80	C:=C & (NOT bit)
ORL C, bit	1001111100000000	9F00	C:=C OR bit
ORL C, /bit	1001111110000000	9F80	C:=C OR (NOT bit)
MOV C, bit	1001110010000000	9C80	(C) := bit
CLR bit	1001110100000000	9D00	(bit) := 0
SET bit	1001110110000000	9D80	(bit) := 1

A.3 Class C

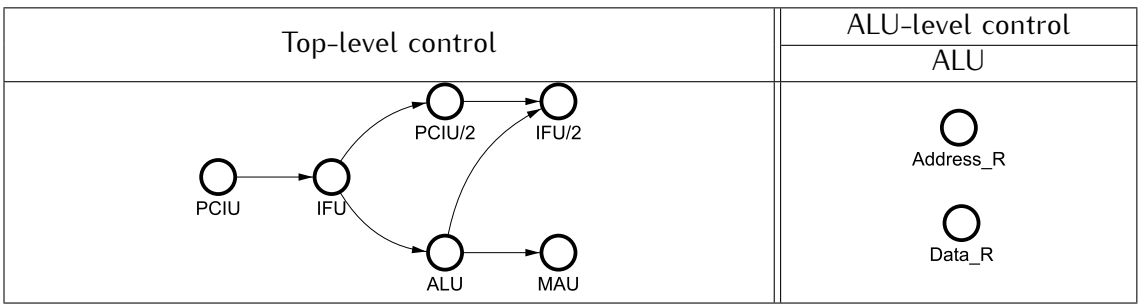


Figure A.3: PO representation for instructions from class C

Table A.3: List of all instructions from class C

Mnemonic	Opcode		Function
	binary	hexadecimal	
MOV A, #data	0010000000000000	2000	(A) := #data
MOV Rn, #data	0010000100010000	2110	(Rn) := #data
MOV DPTR, #data	0010000000100000	2020	(DPTR) := #data
MOV dir, out	0010100000000000	2800	(direct) := #out reading data from the external pins

A.4 Class D

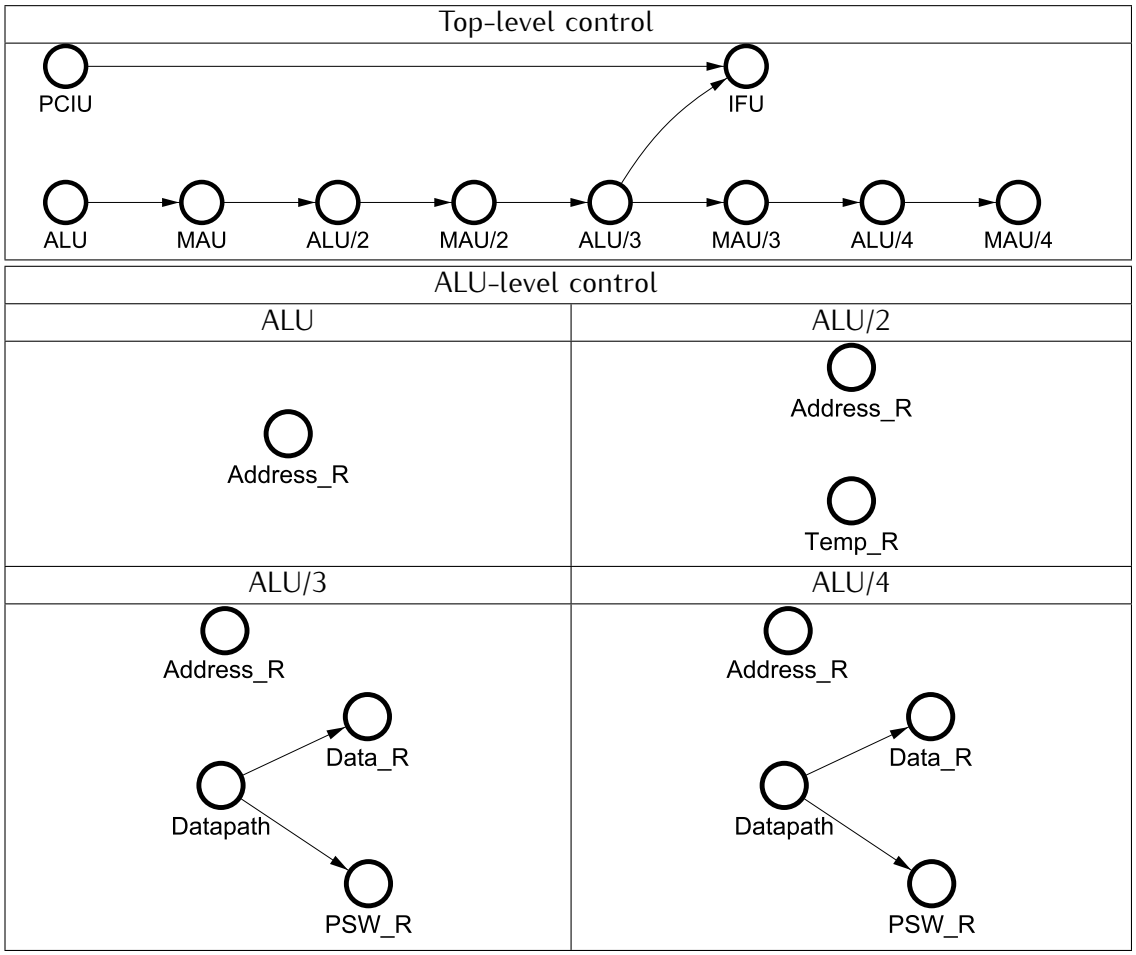


Figure A.4: PO representation for instructions from class D

Table A.4: List of all instructions from class D

Mnemonic	Opcode		Function
	binary	hexadecimal	
MOV @Ri, A	0110000100000010	6102	Move accumulator's content to the internal RAM through the Ri register
MOV A, @Ri	0100000100010000	4110	Move data from the indirect RAM to the accumulator through the Ri register
MOVX @Ri, A	0110000101000010	6142	Move accumulator's content to the external RAM through the Ri register
MOVX A, @Ri	0100000101010000	4150	Move data from the external RAM to the accumulator through the Ri register
MOVX A, @DPTR	0100000010100000	40A0	Move data from the external RAM to the accumulator through the DPTR register
MOVX @DPTR, A	0110000010000100	6084	Move accumulator's content to the external RAM through the DPTR register
ADD A, Rn	0111100100010000	7910	(A) := (A) + (Rn)
ADDC A, Rn	0111100101010000	7950	(A) := (A) + (C) + (Rn)
SUBB A, Rn	0111100110010000	7990	(A) := (A) - (C) - (Rn)
INC @Ri	0100100100010000	4910	Increment the internal RAM by 1 through the Ri register
DEC @Ri	0100100101010000	4950	decrement the internal RAM by 1 through the Ri register
ANL A, Rn	0110100100010000	6910	(A) := (A) & (Rn)
ORL A, Rn	0110100101010000	6950	(A) := (A) OR (Rn)
XRL A, Rn	0110100110010000	6990	(A) := (A) XOR (Rn)
MUL B, A	0111100000001000	7808	$B_{31-16}, A_{15-0} := (A) * (B)$
DIV B, A	0111100001001000	7848	$B_{remainder}, A_{quotient} := (A)/(B)$

A.5 Class E

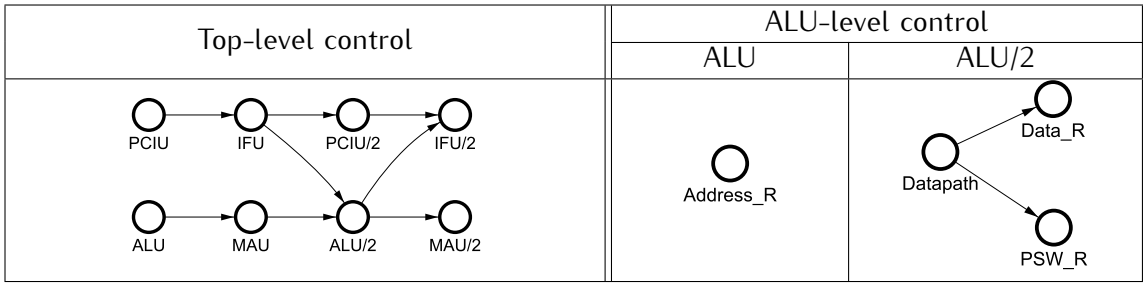


Figure A.5: PO representation for instructions from class E

Table A.5: List of all instructions from class E

Mnemonic	Opcode		Function
	binary	hexadecimal	
MOV dir, A	1100000000000000	C000	(direct) := (A)
MOV dir, Rn	1100000000010000	C010	(direct) := (Rn)
MOV @Ri, #data	1100000010010000	C090	Move immediate data to the internal RAM through the Ri register
ADD A, #data	1100110000000000	CC00	(A) := (A) + #data
ADDC A, #data	1100110100000000	CD00	(A) := (A) + (C) + #data
SUBB A, #data	1100111000000000	CE00	(A) := (A) - (C) - #data
ANL A, #data	1100100000000000	C800	(A) := (A) & #data
ORL A, #data	1100100100000000	C900	(A) := (A) OR #data
XRL A, #data	1100101000000000	CA00	(A) := (A) XOR #data

A.6 Class F

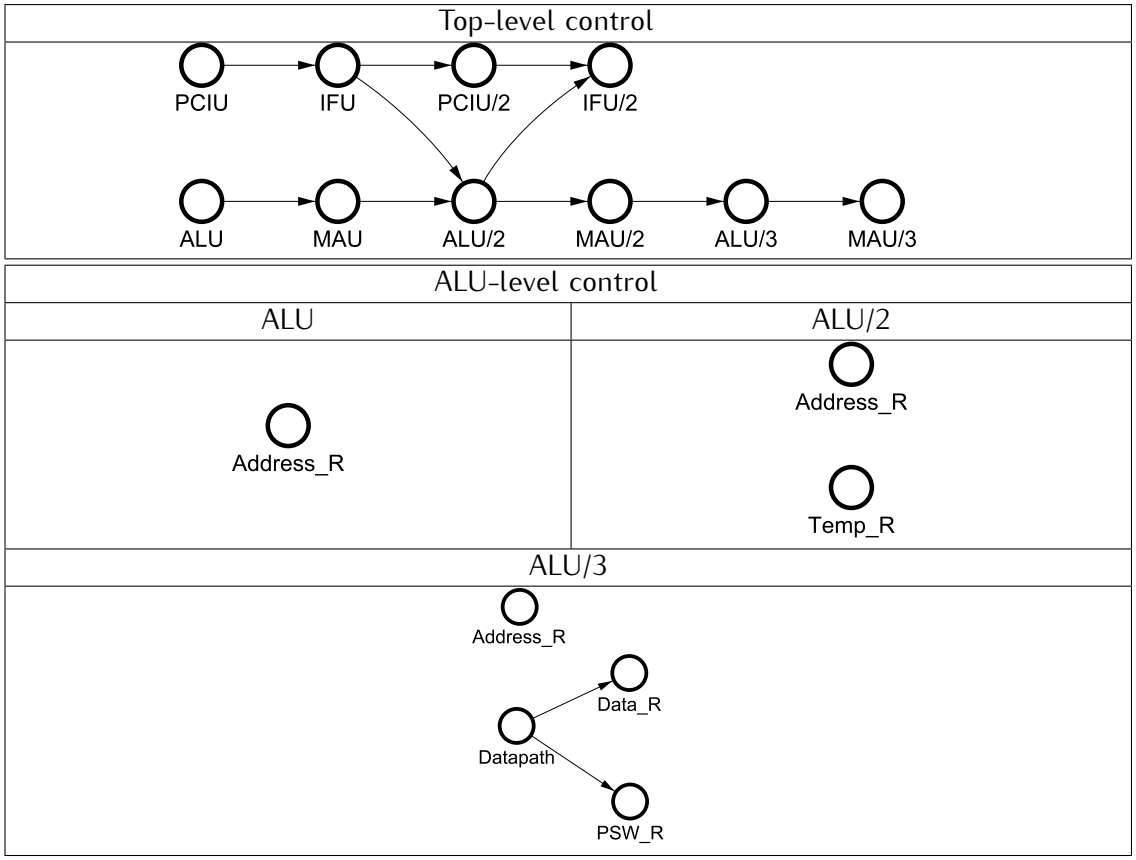


Figure A.6: PO representation for instructions from class F

Table A.6: List of all instructions from class F

Mnemonic	Opcode		Function
	binary	hexadecimal	
MOV @Ri, dir	0001000100010000	1110	Move direct data from one internal RAM location to another one indirectly
ADD A, dir	0001110000000000	1C00	(A) := (A) + (direct)
ADDC A, dir	0001110010000000	1C80	(A) := (A) + (C) + (direct)
SUBB A, dir	0001110100000000	1D00	(A) := (A) - (C) - (direct)
ANL dir, A	0001101000000000	1A00	(direct) := (direct) & (A)
ANL A, dir	0001100000000000	1800	(A) := (A) & (direct)
ORL dir, A	0001101010000000	1A80	(direct) := (direct) OR (A)
ORL A, dir	0001100010000000	1880	(A) := (A) OR (direct)
XRL dir, A	0001101100000000	1B00	(direct) := (direct) XOR (A)
XRL A, dir	0001100100000000	1900	(A) := (A) XOR (direct)

A.7 Class G

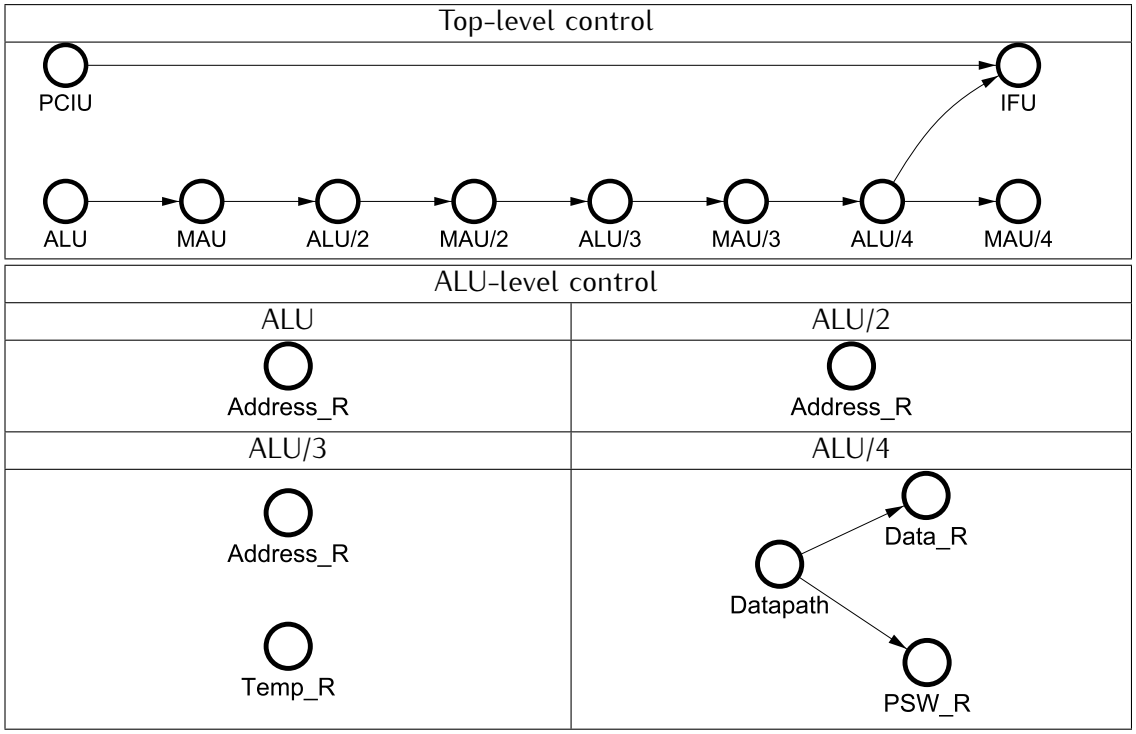


Figure A.7: PO representation for instructions from class G

Table A.7: List of all instructions from class G

Mnemonic	Opcode		Function
	binary	hexadecimal	
ADD A, @Ri	1010100000010000	A810	(A) := (A) + ((Ri))
ADDC A, @Ri	1010101000010000	AA10	(A) := (A) + (C) + ((Ri))
SUBB A, @Ri	1010110000010000	AC10	(A) := (A) - (C) - ((Ri))
ANL A, @Ri	1010000000010000	A010	(A) := (A) & ((Ri))
ORL A, @Ri	1010001000010000	A210	(A) := (A) OR ((Ri))
XRL A, @Ri	1010010000010000	A410	(A) := (A) XOR ((Ri))

All the operations are done with the data from the indirect RAM location (@Ri).

A.8 Class H

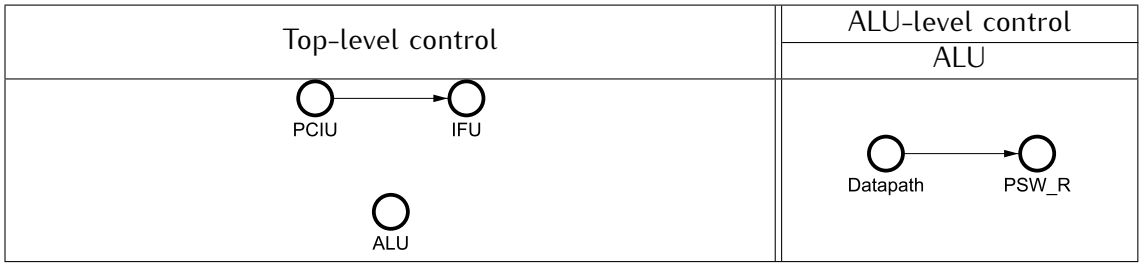


Figure A.8: PO representation for instructions from class H

Table A.8: List of all instructions from class H

Mnemonic	Opcode		Function
	binary	hexadecimal	
CLR C	0000000000000000	0000	(C) := 0
SET C	0000000100000000	0100	(C) := 1
CPL C	0000001000000000	0200	bit C inversion

A.9 Class I

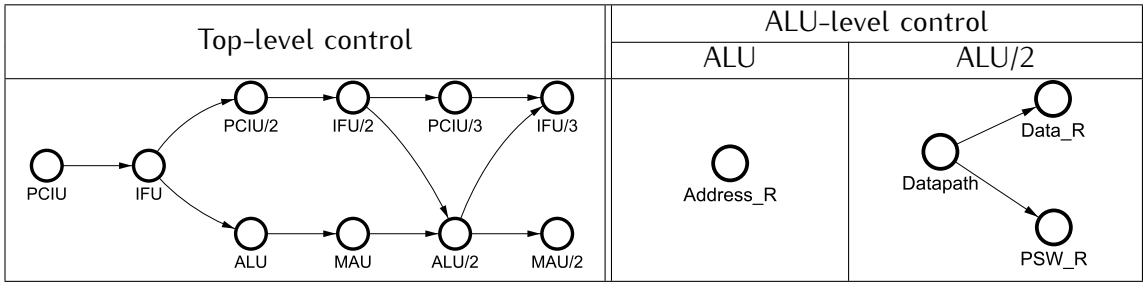


Figure A.9: PO representation for instructions from class I

Table A.9: List of all instructions from class I

Mnemonic	Opcode		Function
	binary	hexadecimal	
MOV dir, dir	1000100000000000	8800	Move direct data from one internal RAM location to another
ANL dir, #data	1000100010000000	8880	(direct) := (direct) & #data
ORL dir, #data	1000100010000100	8884	(direct) := (direct) OR #data
XRL dir, #data	1000100011000000	88C0	(direct) := (direct) XOR #data

A.10 Class J

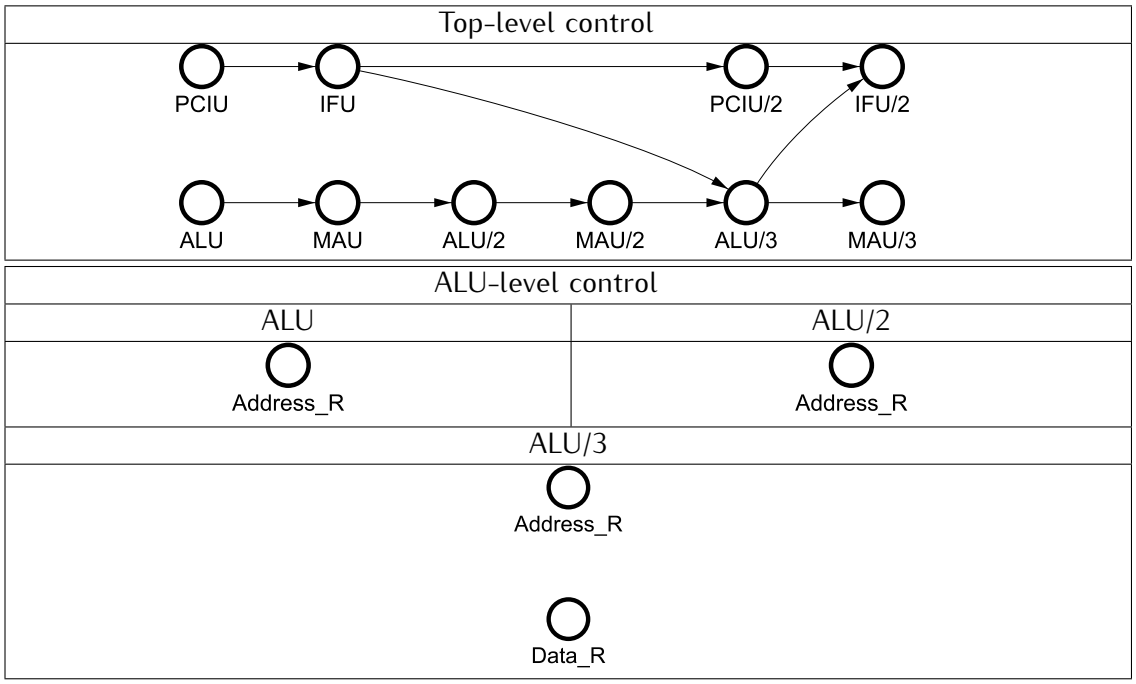


Figure A.10: PO representation for instructions from class J

Table A.10: List of all instructions from class J

Mnemonic	Opcode		Function
	binary	hexadecimal	
MOV dir, @Ri	0011000100010000	3110	Move indirect data from one internal RAM location to another

A.11 Class K

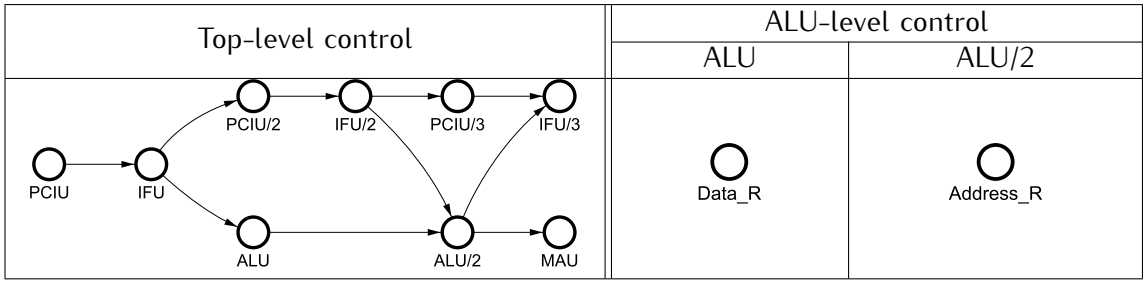


Figure A.11: PO representation for instructions from class K

Table A.11: List of all instructions from class K

Mnemonic	Opcode		Function
	binary	hexadecimal	
MOV dir, #data	1000000000000000	8000	Move immediate data to the internal RAM location

A.12 Class L

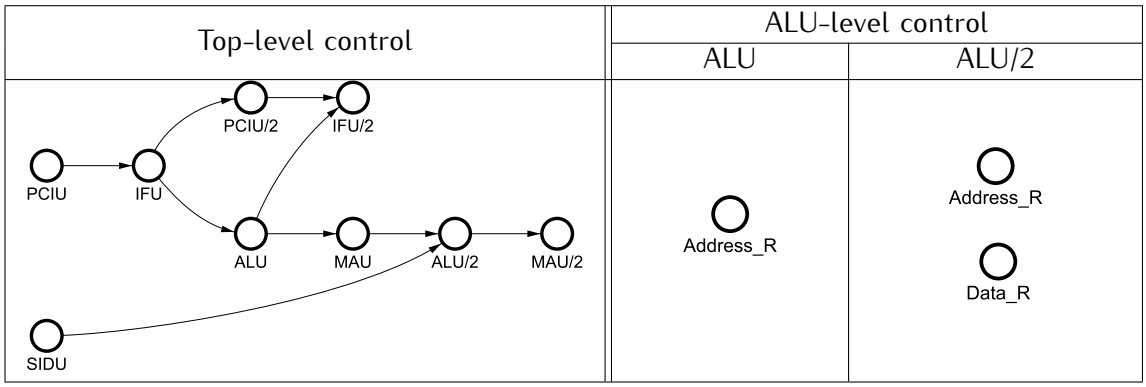


Figure A.12: PO representation for instructions from class L

Table A.12: List of all instructions from class L

Mnemonic	Opcode		Function
	binary	hexadecimal	
PUSH dir	1000100100000011	8903	(SP) := (SP)+1; move data from internal RAM to the Stack

A.13 Class M

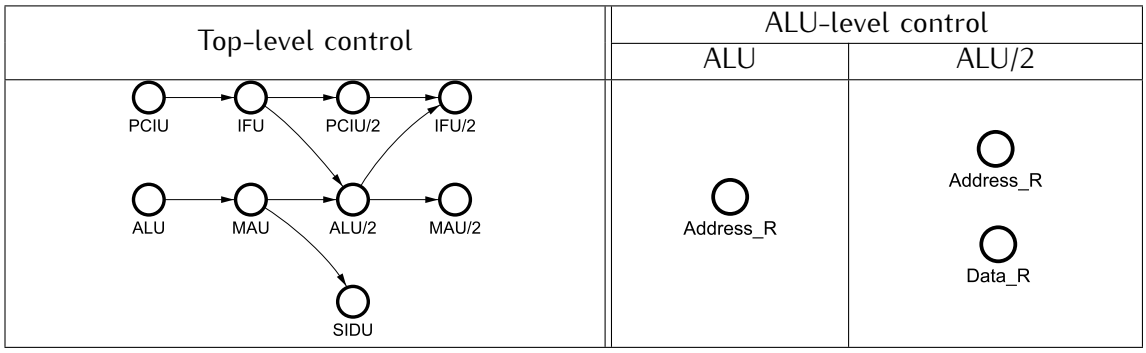


Figure A.13: PO representation for instructions from class M

Table A.13: List of all instructions from class M

Mnemonic	Opcode		Function
	binary	hexadecimal	
POP dir	1000111000011000	8E18	move data from the Stack to internal RAM location; (SP) := (SP)–1

A.14 Class N

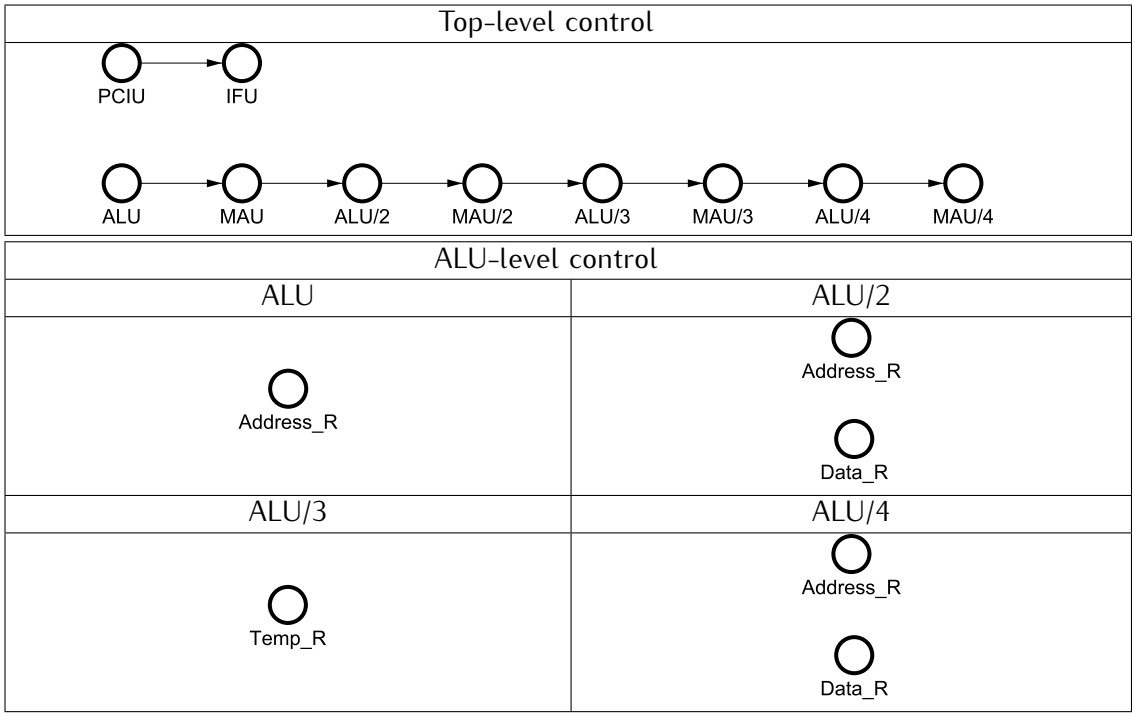


Figure A.14: PO representation for instructions from class N

Table A.14: List of all instructions from class N

Mnemonic	Opcode		Function
	binary	hexadecimal	
XCH A, Rn	1011000000010000	B010	(A) <–> (Rn)

A.15 Class O

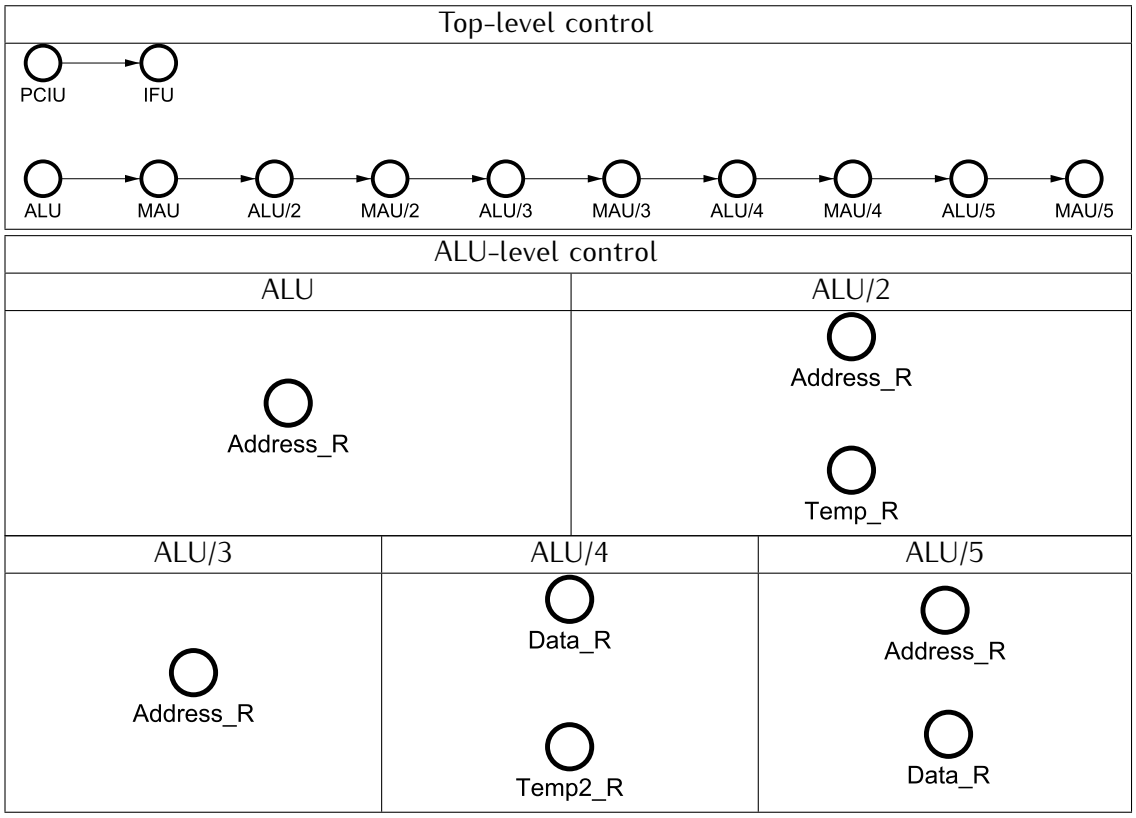


Figure A.15: PO representation for instructions from class O

Table A.15: List of all instructions from class O

Mnemonic	Opcode		Function
	binary	hexadecimal	
XCH A, @Ri	1000010000000010	8402	data exchange between accumulator and indirect RAM location
XCHD A, @Ri	1000010010000010	8482	half-word data exchange between accumulator and indirect RAM location

A.16 Class P

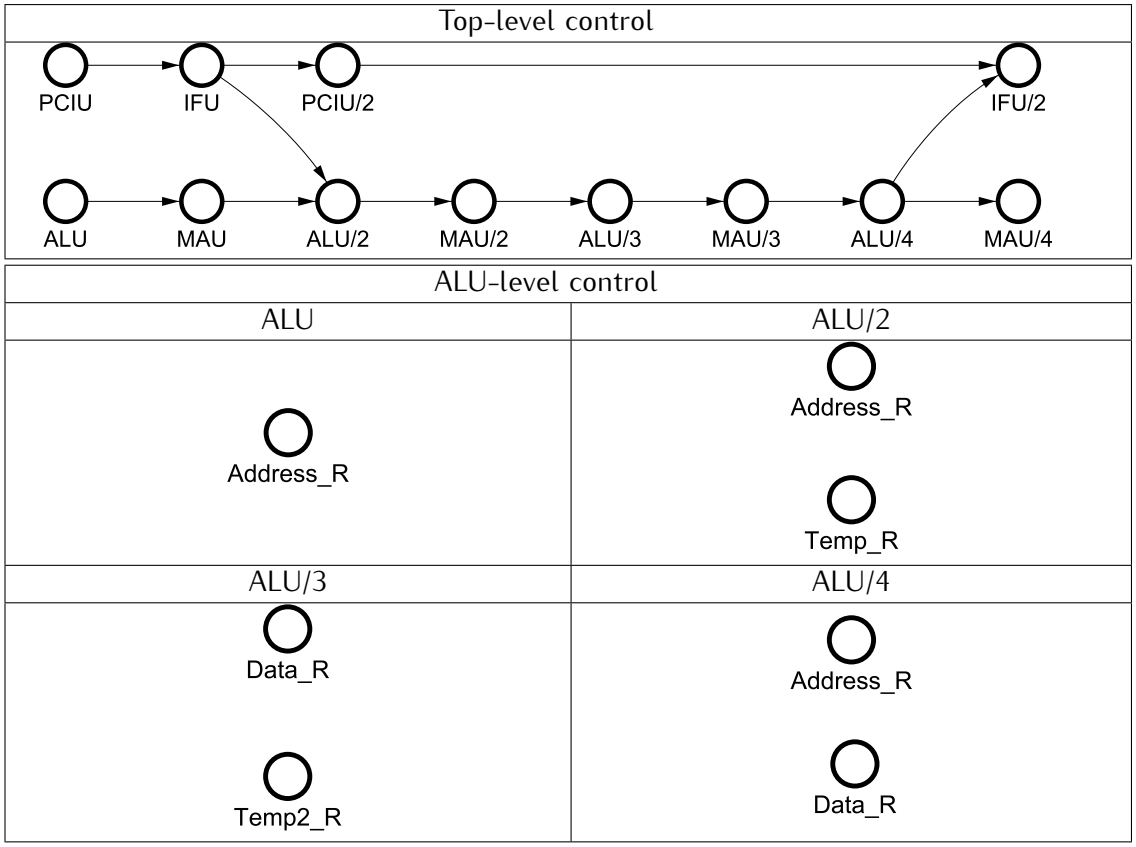


Figure A.16: PO representation for instructions from class P

Table A.16: List of all instructions from class P

Mnemonic	Opcode		Function
	binary	hexadecimal	
XCH A, dir	1000111100000000	8F00	(A) <–> (dir)

A.17 Class Q

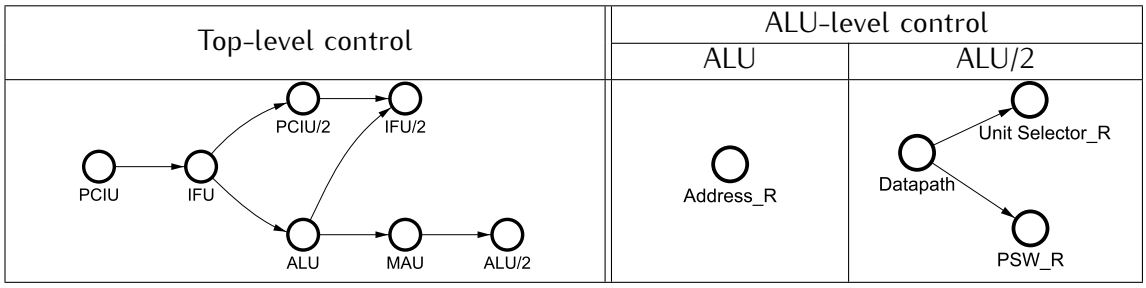


Figure A.17: PO representation for instructions from class Q

Table A.17: List of all instructions from class Q

Mnemonic	Opcode		Function
	binary	hexadecimal	
MOV PSW, dir	0011001000000000	3200	move data from internal RAM location to PSW
MOV C, bit	0011001100000000	3300	(C) := bit
MOV wrk, dir	0011001110000000	3800	move data from internal RAM location to Unit Selector register

A.18 Class R

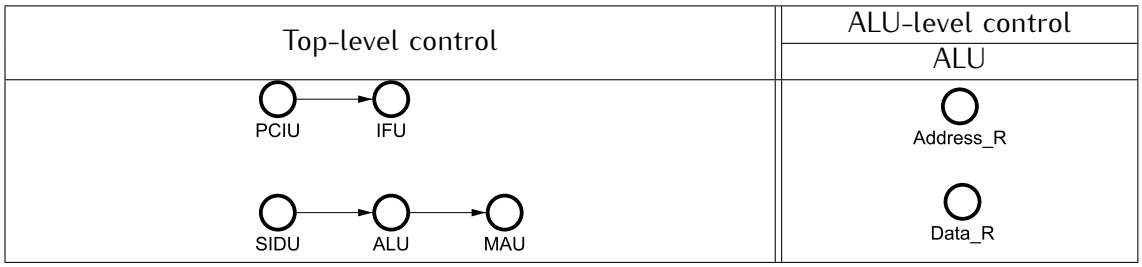


Figure A.18: PO representation for instructions from class R

Table A.18: List of all instructions from class R

Mnemonic	Opcode		Function
	binary	hexadecimal	
PUSH PSW	1000110000011000	8C18	(SP) := (SP)+1; move data from the PSW register to the Stack

A.19 Class S

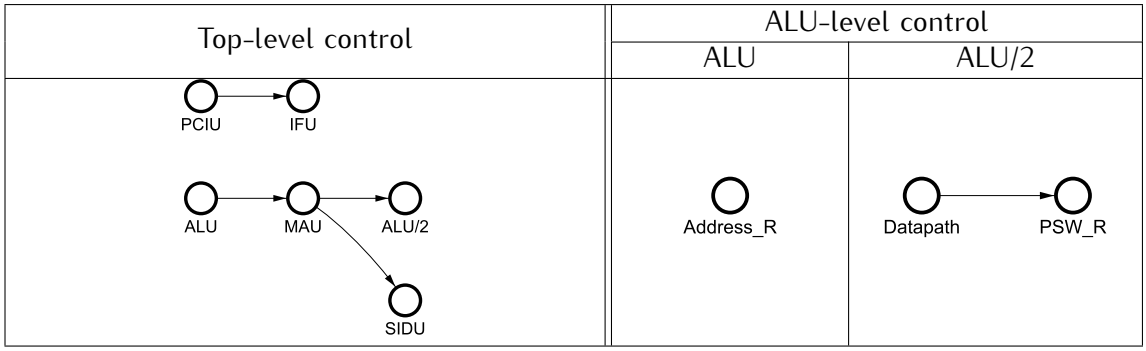


Figure A.19: PO representation for instructions from class S

Table A.19: List of all instructions from class S

Mnemonic	Opcode		Function
	binary	hexadecimal	
POP PSW	1000110101011100	8D5C	move data from the Stack to the PSW register; (SP) := (SP)–1

A.20 Class T

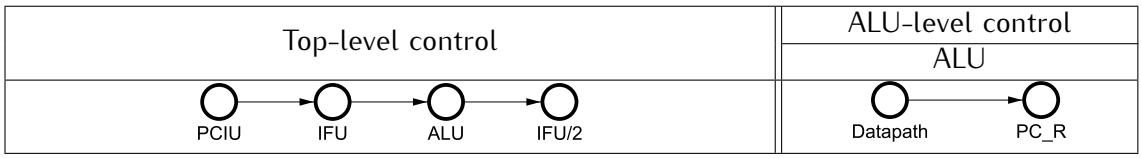


Figure A.20: PO representation for instructions from class T

Table A.20: List of all instructions from class T

Mnemonic	Opcode		Function
	binary	hexadecimal	
LJMP addr	1101100000000000	D800	(PC):=addr
SJMP rel	1101110000000000	DC00	(PC):=(PC) + rel

A.21 Class U

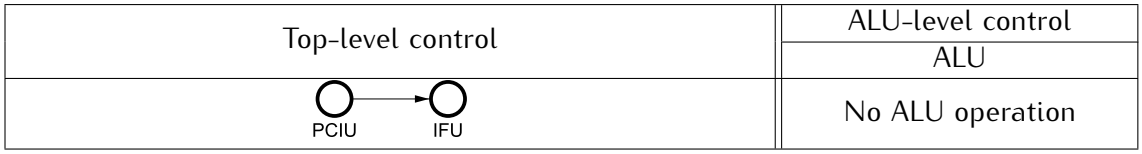


Figure A.21: PO representation for instructions from class U

Table A.21: List of all instructions from class U

Mnemonic	Opcode		Function
	binary	hexadecimal	
NOP	0000011100000000	0700	(PC) := (PC) + 1

A.22 Class V

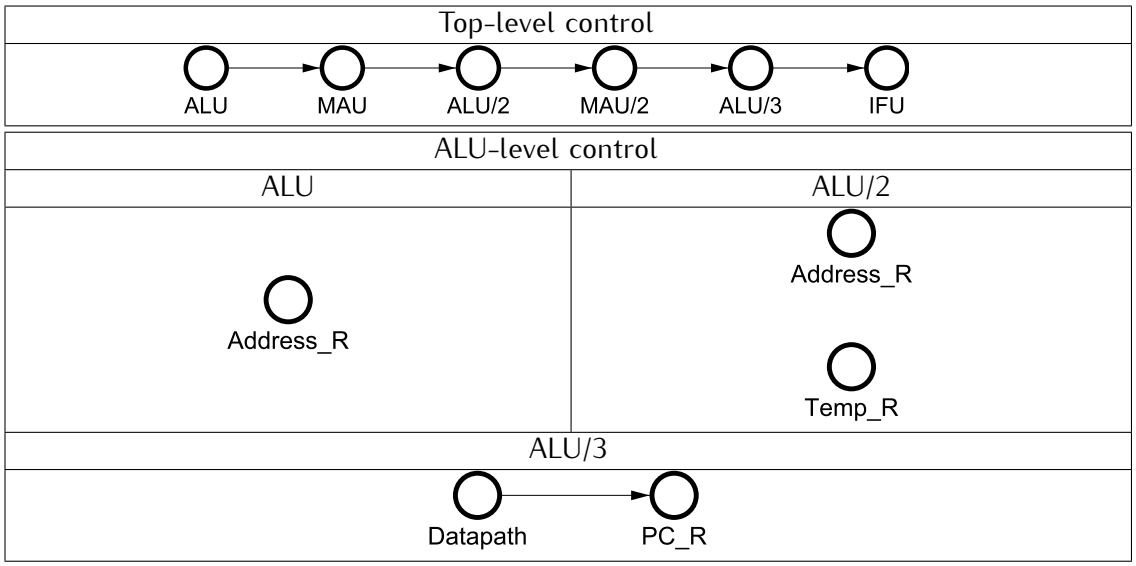


Figure A.22: PO representation for instructions from class V

Table A.22: List of all instructions from class V

Mnemonic	Opcode		Function
	binary	hexadecimal	
JMP @A+DPTR	0000011000100000	0620	(PC):=(A) + (DPTR)

A.23 Class W

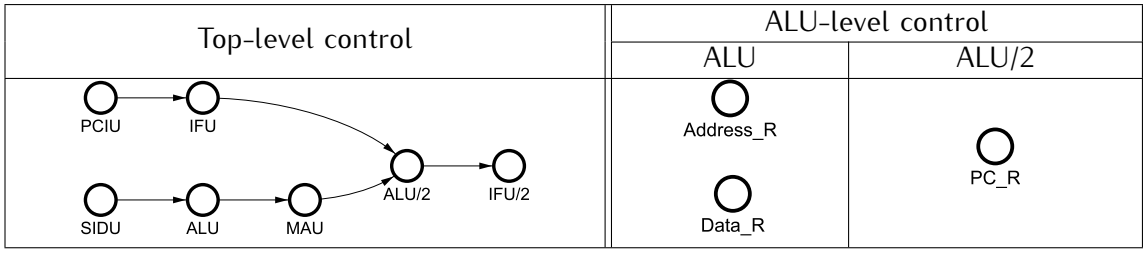


Figure A.23: PO representation for instructions from class W

Table A.23: List of all instructions from class W

Mnemonic	Opcode		Function
	binary	hexadecimal	
LCALL addr	1101000000011000	D018	(SP) := (SP)+1; move PC to the Stack; (PC):=addr

A.24 Class X

Figure A.24: PO representation for instructions from class X

Table A.24: List of all instructions from class X

Mnemonic	Opcode		Function
	binary	hexadecimal	
LCALL addr	1101000000011000	D018	(SP) := (SP)+1; move PC to the Stack; (PC):=addr

A.25 Class Y

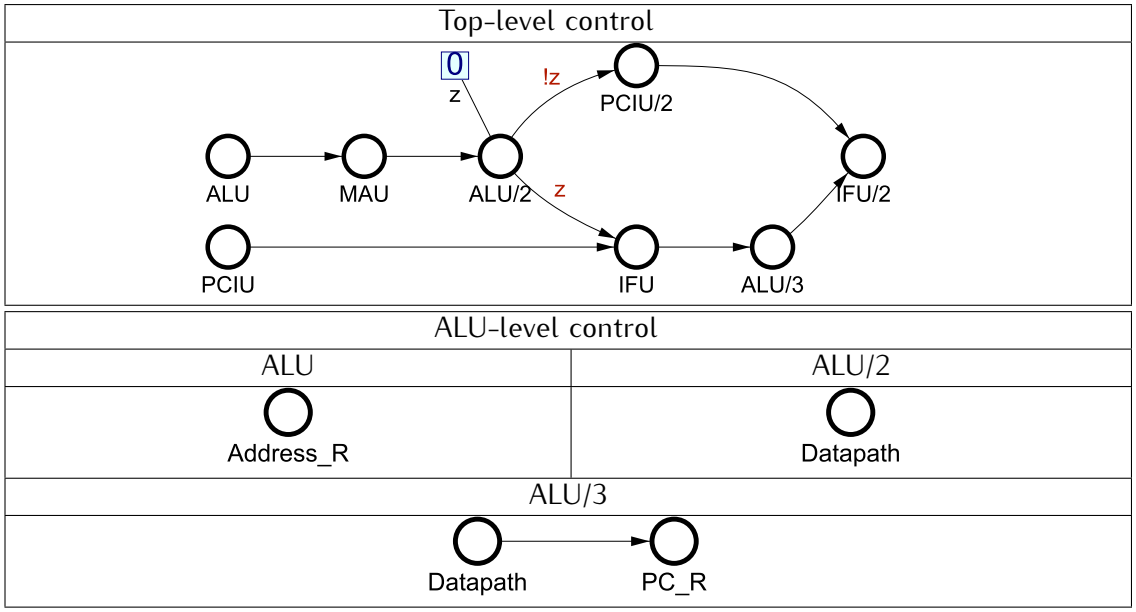


Figure A.25: PO representation for instructions from class Y

Table A.25: List of all instructions from class Y

Mnemonic	Opcode		Function
	binary	hexadecimal	
JZ rel	1000101100000000	8B00	(PC):=(PC) + 1, if (A) = 0 then (PC):=(PC) + rel

A.26 Class Z

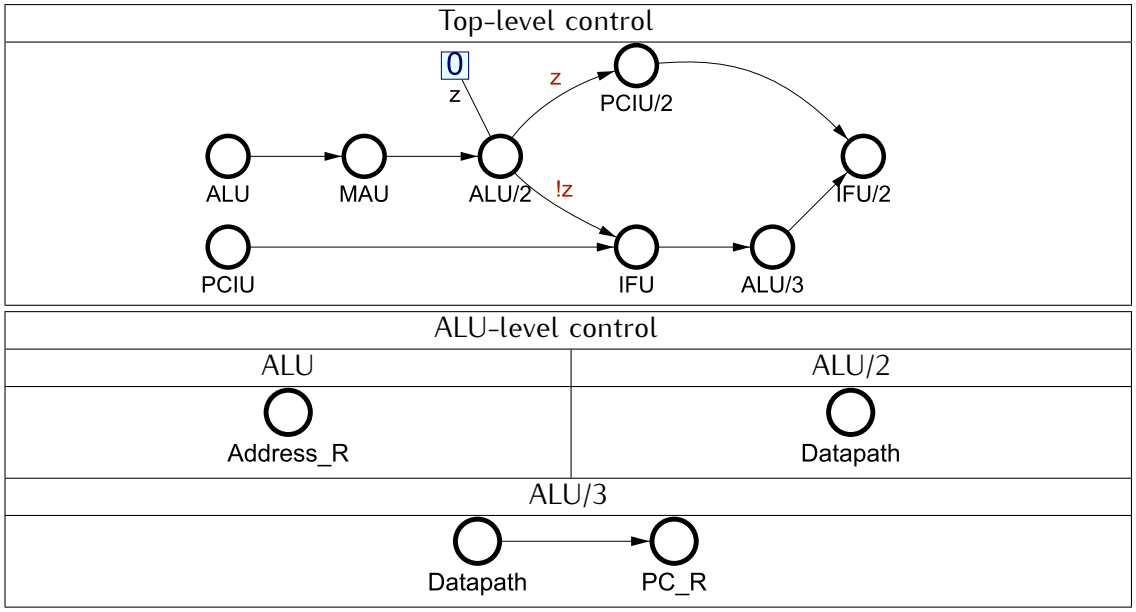


Figure A.26: PO representation for instructions from class Z

Table A.26: List of all instructions from class Z

Mnemonic	Opcode		Function
	binary	hexadecimal	
JNZ rel	0011011000000000	3600	(PC):=(PC) + 1, if (A)<>0 then (PC):=(PC) + rel

A.27 Class AA

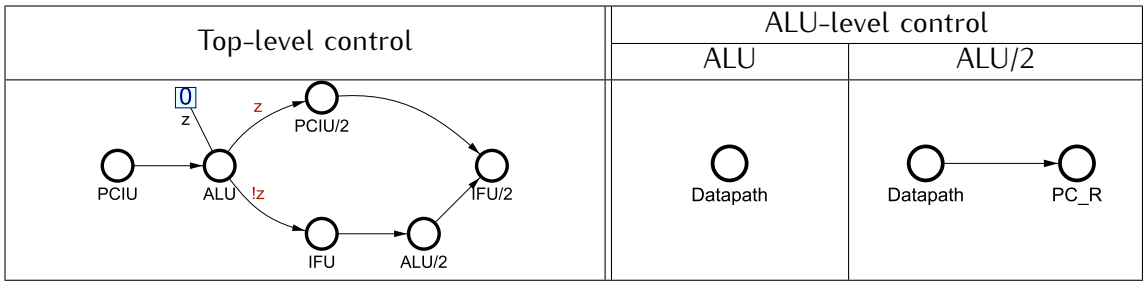


Figure A.27: PO representation for instructions from class AA

Table A.27: List of all instructions from class AA

Mnemonic	Opcode		Function
	binary	hexadecimal	
JC rel	0011010000000000	3400	(PC):=(PC) + 1, if (C) = 1 then (PC):=(PC) + rel

A.28 Class AB

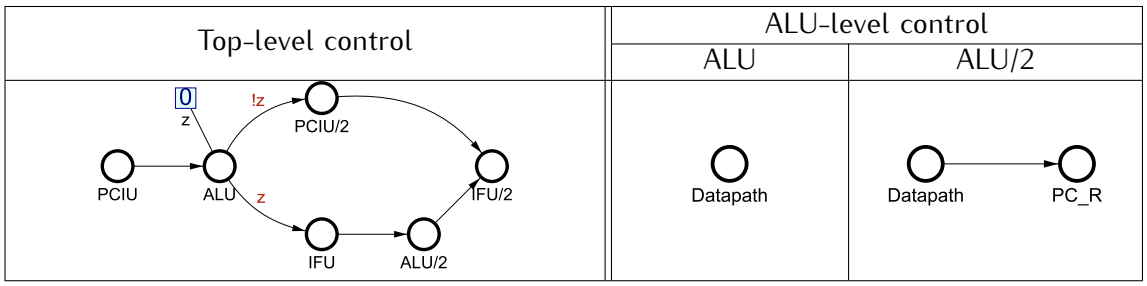


Figure A.28: PO representation for instructions from class AB

Table A.28: List of all instructions from class AB

Mnemonic	Opcode		Function
	binary	hexadecimal	
JNC rel	0011010000000000	3400	(PC):=(PC) + 1, if (C) = 0 then (PC):=(PC) + rel

A.29 Class AC

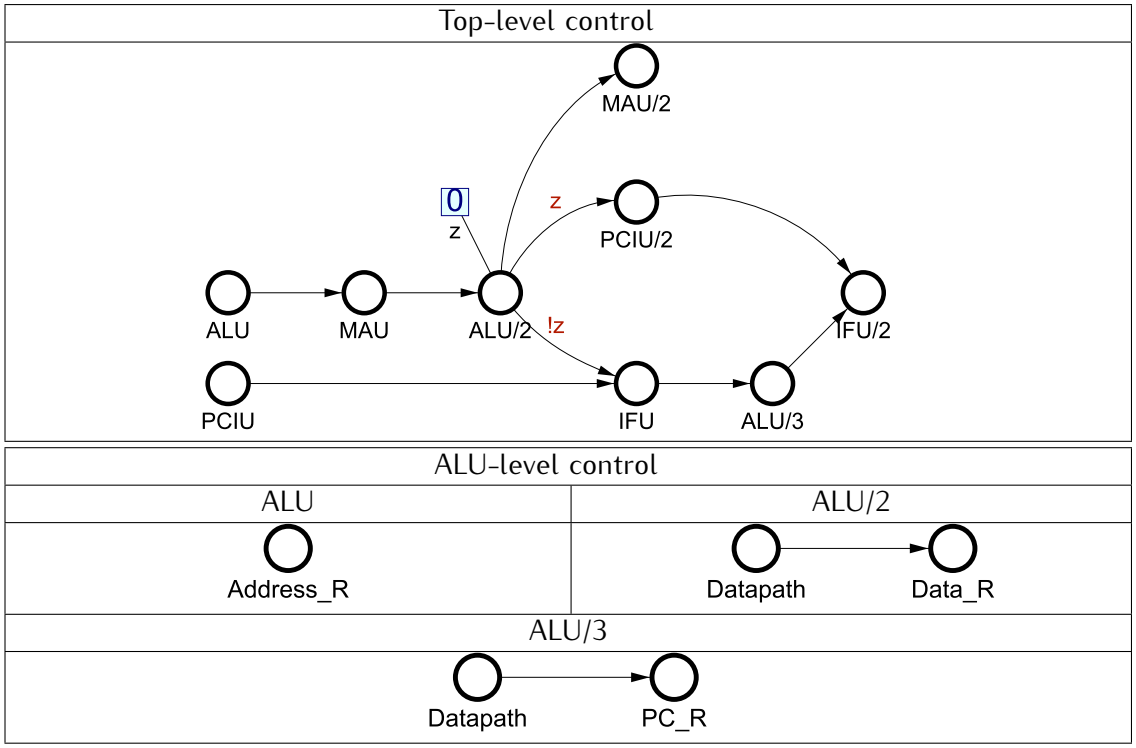


Figure A.29: PO representation for instructions from class AC

Table A.29: List of all instructions from class AC

Mnemonic	Opcode		Function
	binary	hexadecimal	
DJNZ Rn, rel	0011110100010000	3D10	(PC):=(PC) + 1; (Rn):=(Rn) - 1 if (Rn) <> 0 then (PC):=(PC) + rel

A.30 Class AD

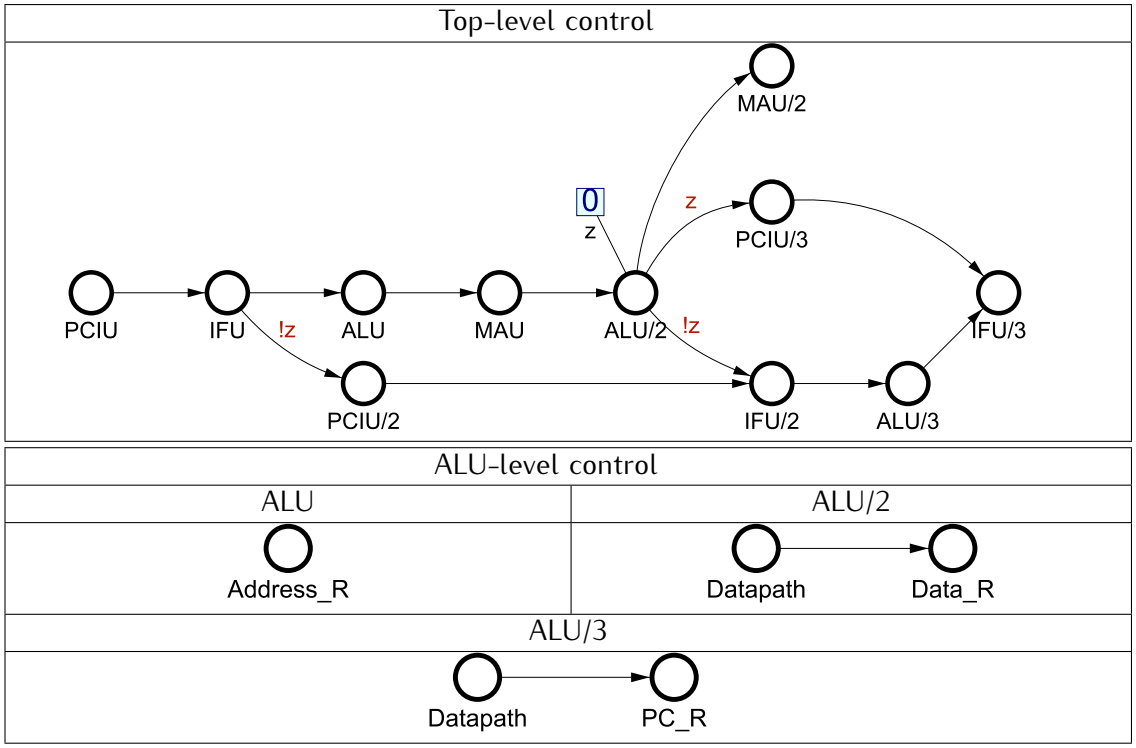


Figure A.30: PO representation for instructions from class AD

Table A.30: List of all instructions from class AD

Mnemonic	Opcode		Function
	binary	hexadecimal	
DJNZ dir, rel	0011010100000000	3500	(PC):=(PC) + 1; (dir):=(dir) - 1 if (Rn) <> 0 then (PC):=(PC) + rel

A.31 Class AE

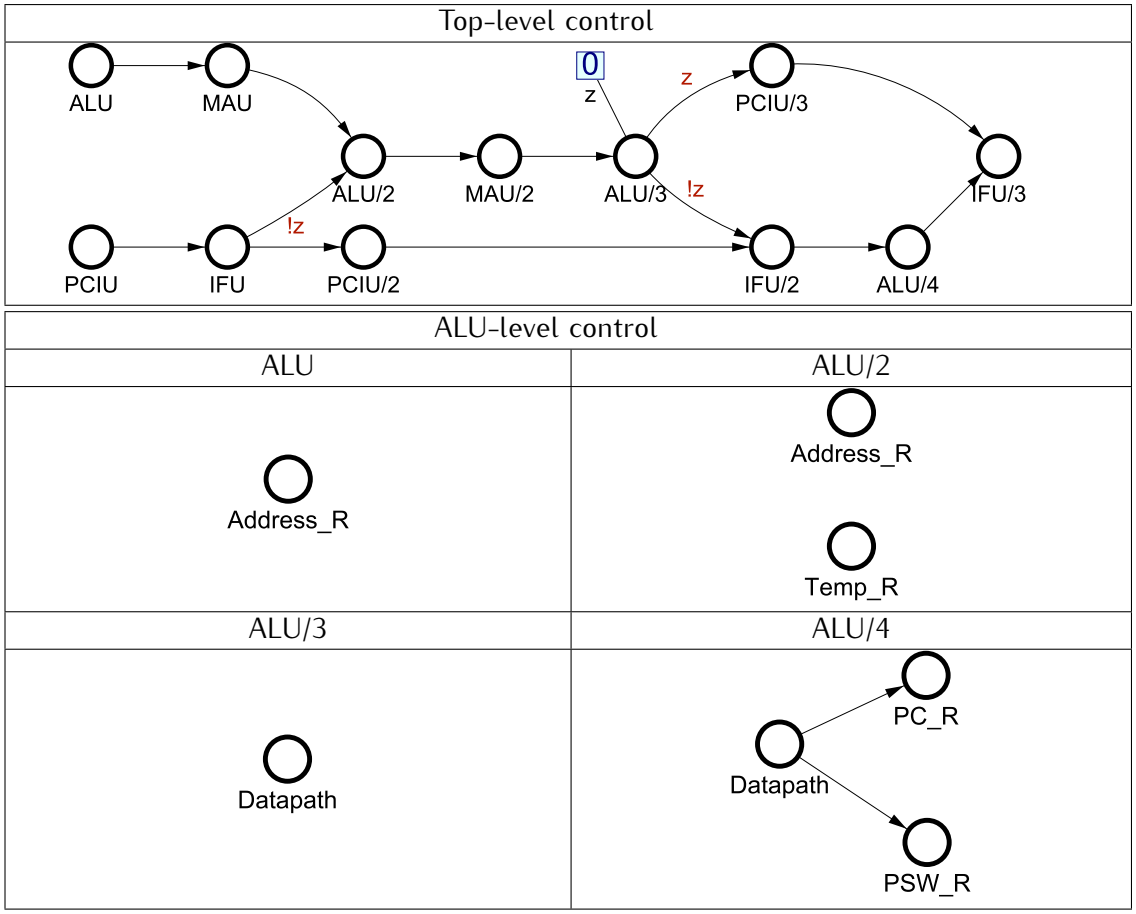


Figure A.31: PO representation for instructions from class AE

Table A.31: List of all instructions from class AE

Mnemonic	Opcode		Function
	binary	hexadecimal	
CJNE @Ri, #data, rel	0000101000010000	0A10	(PC):=(PC) + 2, if indirect data in RAM <> #data then (PC):=(PC) + rel if the data < #data then (C):=1 else (C):=0
CJNE A, dir, rel	0000100000000000	0800	(PC):=(PC) + 2 if (A) <> direct data from internal RAM then (PC):=(PC) + rel if (A) < (direct) then (C):=1 else (C):=0

A.32 Class AF

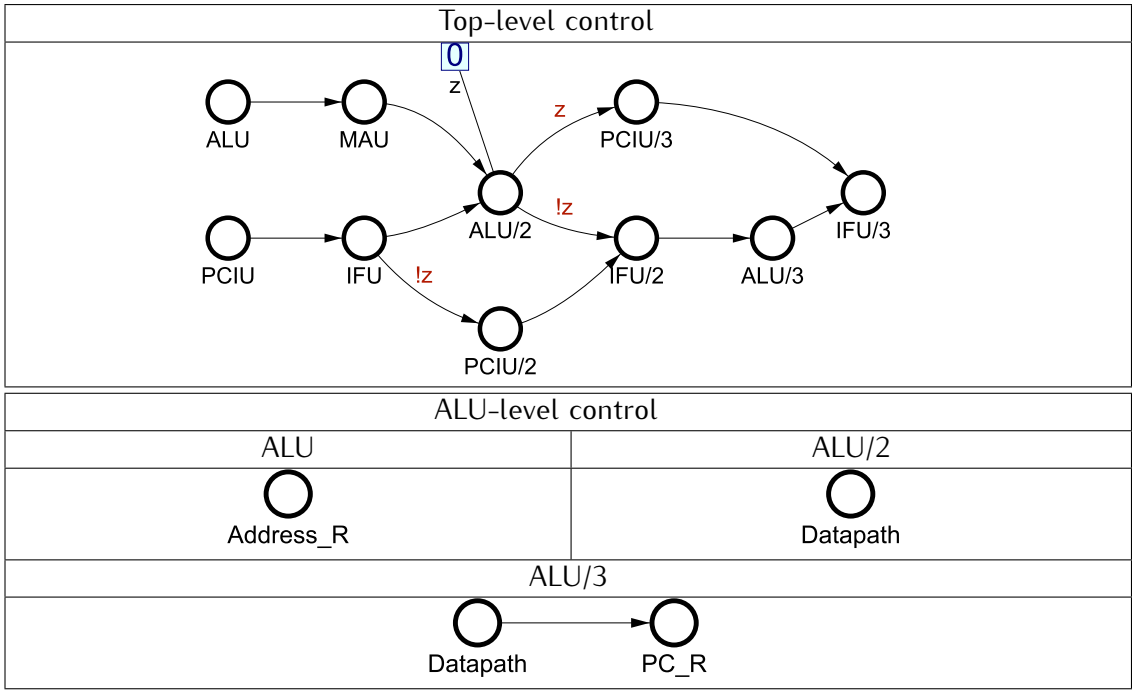


Figure A.32: PO representation for instructions from class AF

Table A.32: List of all instructions from class AF

Mnemonic	Opcode		Function
	binary	hexadecimal	
CJNE Rn, #data, rel	1011100000010000	B810	(PC):=(PC) + 2 if (Rn)<> #data then (PC):=(PC) + rel if (Rn) < #data then (C):=1 else (C):=0
CJNE A, #data, rel	1011100000000000	B800	(PC):=(PC) + 2 if (A) <> #data then (PC):=(PC) + rel if (A) < #data then (C):=1 else (C):=0

A.33 Class AG

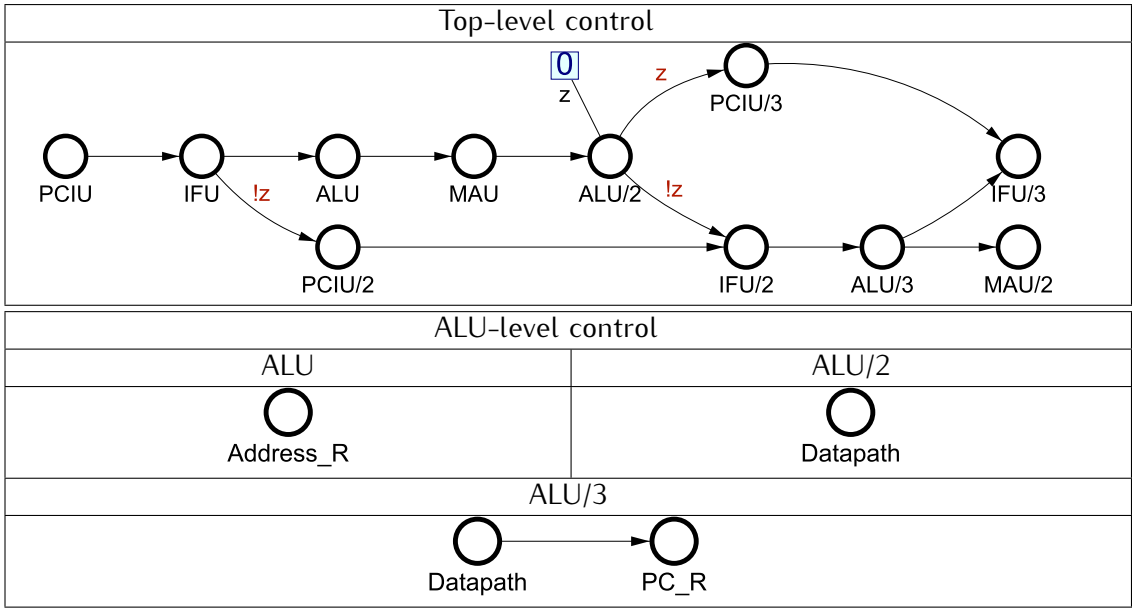


Figure A.33: PO representation for instructions from class AG

Table A.33: List of all instructions from class AG

Mnemonic	Opcode		Function
	binary	hexadecimal	
JB bit, rel	1000101000000000	8A00	(PC) := (PC) + 2 if (bit) = 1 then (PC) := (PC) + rel

A.34 Class AH

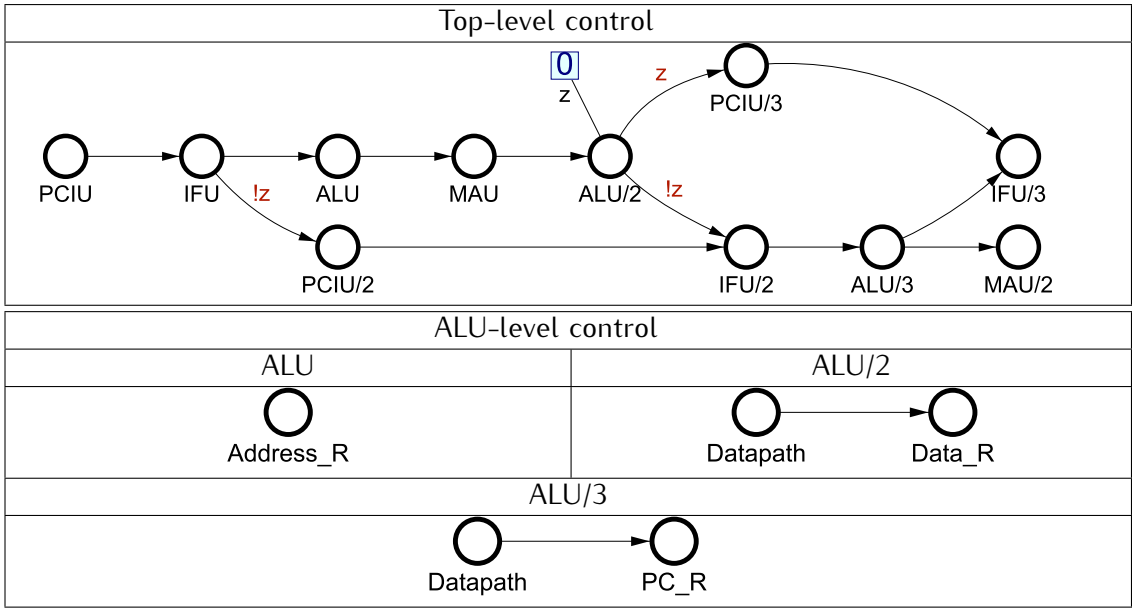


Figure A.34: PO representation for instructions from class AH

Table A.34: List of all instructions from class AH

Mnemonic	Opcode		Function
	binary	hexadecimal	
JBC bit, rel	0000110000000000	0C00	(PC) := (PC) + 2 if (bit) = 1 then (bit) := 0, (PC) := (PC) + rel

A.35 Class AI

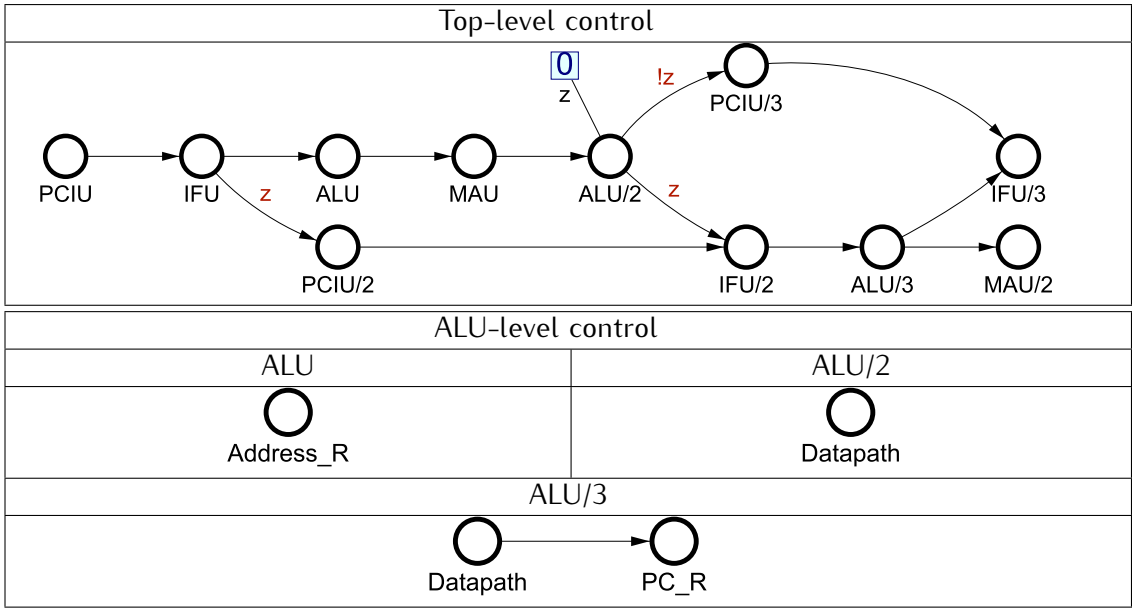


Figure A.35: PO representation for instructions from class AI

Table A.35: List of all instructions from class AI

Mnemonic	Opcode		Function
	binary	hexadecimal	
JNB bit, rel	0000110100000000	0D00	(PC) := (PC) + 2 if (bit) = 0 then (PC) := (PC) + rel

A.36 Class AJ

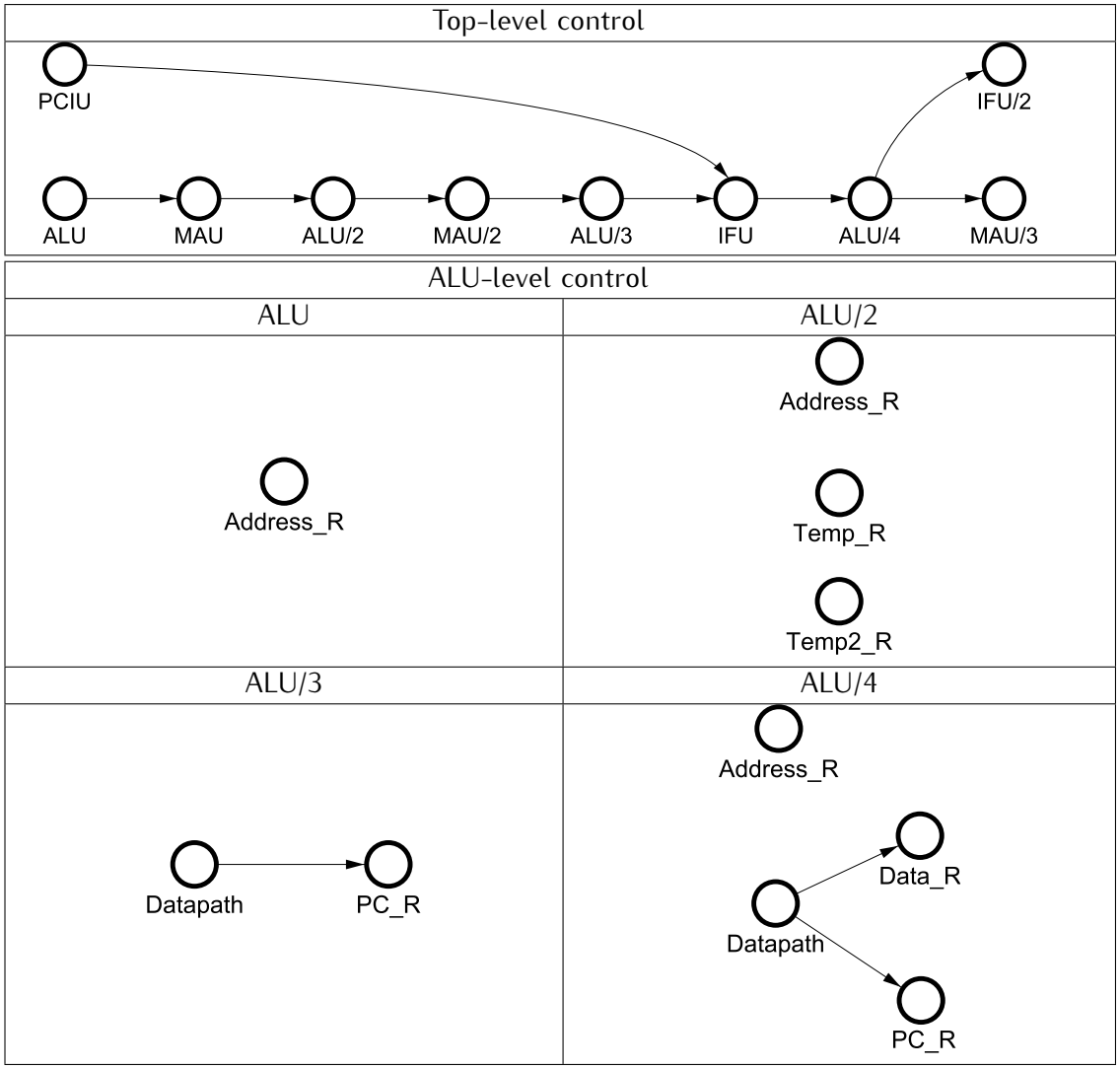


Figure A.36: PO representation for instructions from class AJ

Table A.36: List of all instructions from class AJ

Mnemonic	Opcode		Function
	binary	hexadecimal	
MOVC A, @A+DPTR	0000111000100000	0E20	Move the code data relative to the DPTR to the accumulator (address=A+DPTR)

A.37 Class AK

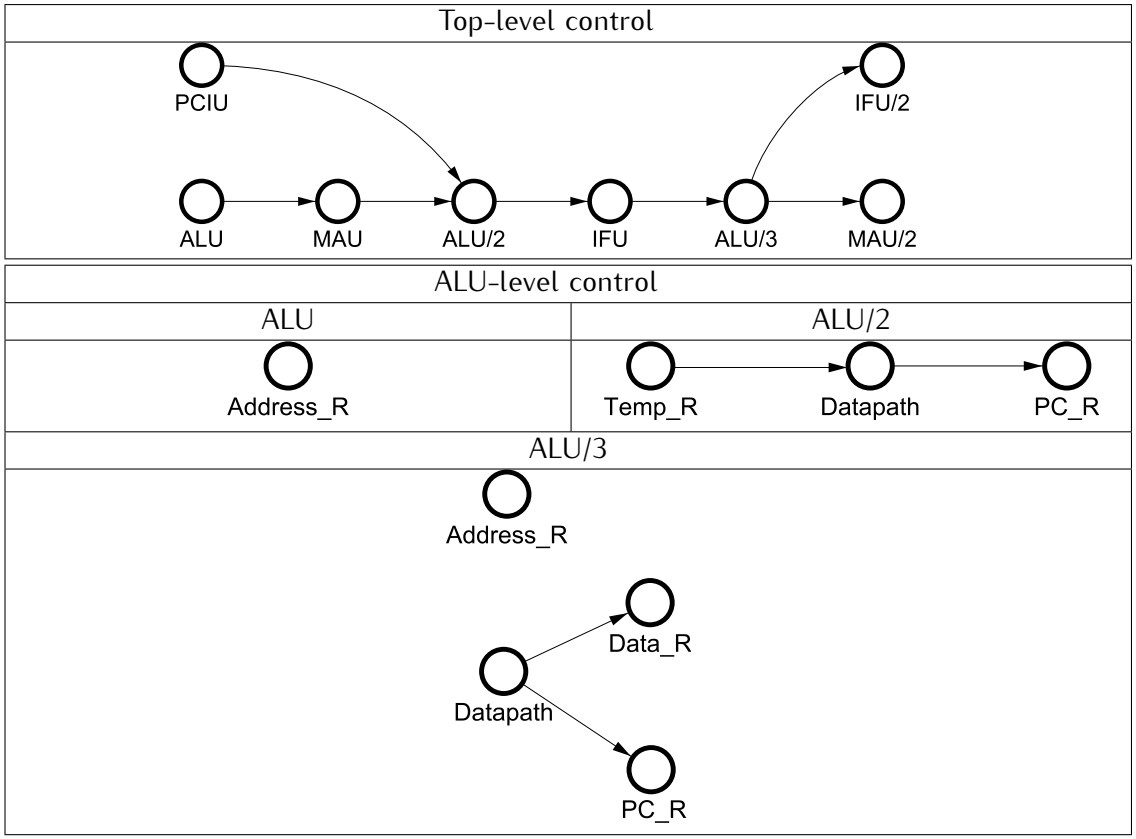


Figure A.37: PO representation for instructions from class AK

Table A.37: List of all instructions from class AK

Mnemonic	Opcode		Function
	binary	hexadecimal	
MOVC A, @A+PC	0000111100000000	0F00	Move the code data relative to the PC to the accumulator (address=A+PC)

A.38 Interrupt

Section 4.3.7 explains the order of activation of functional units in the situation when a processor interrupt occurs. There are two requests (ALU/6 and ALU/7) to the ALU block. The following table addresses these two requests:

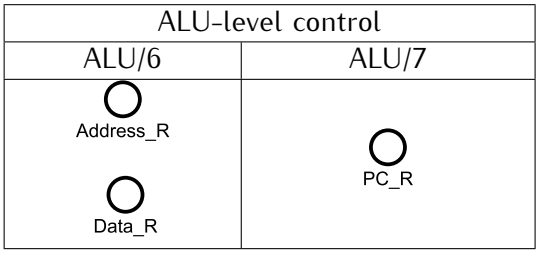


Figure A.38: PO representation for the interrupt handler

Appendix B

Boolean equations for microcontroller synthesis

This appendix presents the resultant Boolean equations from the mapping stage in Section 4.2.1. As we have two control logics (the Top-level and the ALU control), we separate this appendix also into two Sections.

B.1 Boolean equations for the Top-level microcontroller

```
req_sidu <= go and not C and ((not E and D and B and A) or (not D
and not B and ((not G and H and not F and E and A) or (F and
((not G and not E and  not A and ackmau) or (E and A and
((ackmau and (not H or not G)) or (not G and not H))))))));

req_alu <= go and ((not E and B and acksidu) or (ackifu and ((not
B and ((not E and A) or (not G and not F) or (G and not H))) or
(C and ackpciu) or (E and D))) or (not D and ((G and H and E and
not C) or (F and ((not C and ((not G and not E) or (G and not H)))
or (A and (acksidu or H)))) or B)) or (not A and ((D and ((F and
((ackpciu and (not H or G)) or (G and not H))) or (not G and
not F) or not C or E)) or (E and ackifu) or (not F and  not C)
or B)) or (C and A));
```

```
req_alu2 <= go and ackalu and ((not E and  not B and ((F and D and C and
not A and ackifu and ((not G and not H and not z) or (G and H and z)))
or (not F and not D and not C and A and ackifu2))) or (ackmau and
((C and ((A and (not D or not E)) or B)) or (ackifu and ((D and
((not E and ((A and acksidu) or (F and not C))) or (E and C)))
or (not D and B))) or (not A and ((D and C and ((G and not H) or E
or not F)) or (not C and ((F and not D and ((E and ackpciu) or not
H)) or (not F and E and ackifu))) or B)) or (not B and ((not D and
((not G and H and F and not C) or (A and ((not F and ((E and ((H and
acksidu) or G)) or (not H and ackifu2))) or (F and not E))))) or
(ackifu and ((D and ((not G and H) or not F)) or (not C and ((E and
((H and F and ackpciu) or (G and not H))) or (not H and D)))))))));

req_pciu <= go and ((not A and ((not C and (D or not F)) or B))
or (G and H) or A or C or E);

req_ifu <= go and ((not E and not D and not A and F and not C and not B
and ackalu2 and ((not H and ackalu3) or not G))or (ackpciu and ((A
and ((B and (ackalu2 or not C)) or (F and  not C))) or (ackalu3
and ((not D and not A and ackalu2) or (not C and ((E and not G) or D))
or (B and ((E and not F and not C) or (ackalu2 and ((H and not G) or
ackalu or not F)))))) or (not B and ((not A and ((not z and ackalu2
and ((H and not G) or not F)) or (not E and not F))) or (not C and ((H
and ((F and ackalu2) or (not E and G))) or (not F and ((z and ackalu2)
or not G or not A or not H)) or (E and not G) or D)) or (C and ((not
D and ackalu4 and (ackalu2 or F or not G or not H or not E)) or
(not A and ((not E and H and not G) or (ackalu and ((z and not E and H)
or (not z and ((ackalu2 and (not H or E)) or (not E and not G))))) or
not D)))) or (D and A))))));

req_mau <= go and ackalu and ((not A and not C and D) or (not B and ((D
and not F) or (A and F))) or (E and (not D or C)) or (not E and ((F and
((not H and (G or not C)) or (not G and H))) or (A and ackalu2) or B))
or (C and not D));
```

```
req_mau2 <= go and ackalu2 and ((not E and not B and ((D and((not
 G and (H or not F)) or A)) or (A and F))) or (not A and E and
 D) or (not D and ((A and ((F and G) or C)) or B))or (not C and
 ((not D and F and G and not H) or (D and ((notB and (not G or
 not F)) or not A)) or (E and ((not F and ((not B and not G)
 or not A)) or (ackalu3 and ((not A and not H and not z) or
 (not D and F and G)))))))) or (C and B));

req_alu3 <= go and ((not A and ackmau2 and ((not C and ((not H
and not D and ((not F and E) or (F and G))) or (ackifu and ((not
 F and E) or D)))) or B)) or (not B and ((A and F and not E and
 not D and ackmau2) or (C and ((not A and F and not z and D and
 ackifu and ((not G and H and ackifu2) or (G and not H))) or
 (ackmau2 and ((not E and ((not F and not G and D and ackifu)
 or A)) or (A and not D))))) or (E and ((C and not z and D and
 ((A and ackifu2) or (not A and ackifu))) or (not C and not D
 and ((A and not F and G and ((not H and not z and ackifu2)
 or (H and z and ackifu))) or (F and ((not A and not G and not
 H and not z and ackifu2) or (H and ackifu and ((not A and z
 and ackifu2) or (G and (ackmau2 or not A))))))))))));

req_pciu2 <= go and (ackpciu and ((A and not C and not D and B and
 ackifu) or (not B and ((A and not C and not D and E and G and H
 and not F and not z and ackalu2) or (not A and C and D and ((F
 and ackalu and ((not E and G and H and not z) or (not H and z
 and ((not E and not G) or ackalu2)))) or (E and z and ackalu2)))
 or (ackifu and ((not A and C and not D) or (not E and ((not A
 and D and not G and H) or (not F and ((not A and D) or (A and
 not C))))) or (E and ((not A and not D and not G) or (not F and
 ((not C and not G) or (not A and not D))) or (A and ((not C and
 G and (F or not H)) or D)))) or (not C and D)))))));
```

```
req_ifu2 <= go and ((A and not C and ((E and B and D and ackalu) or
(ackalu2 and ((B and D and ackalu) or (ackpciu2 and ((ackalu and
((not F and not z) or B)) or (not D and ((not H and ((E and G and
(not z or F)) or B)) or (not E and B)))))))) or (not B and ((not C
and not D and ((A and not E and not F and ackpciu2) or (not H and
((A and not G and not F and ackpciu2) or (E and ackalu2 and ((not A
and G and F and ackalu4) or (not z and ackpciu2 and ((not A and not
G and F) or (not F and ackalu3)))))))))) or (ackalu and ((ackalu2 and
((A and not C and E and G and H and not F and z and not D and
ackalu3) or (not A and ((not C and E and G and H and F and not D and
ackalu3) or (C and D and ((not E and G and H and F and z) or (not z
and ((not E and not G and not H and F) or (ackalu3 and ((not H and F)
or E)))))))))) or (ackpciu2 and ((A and E and not z and D and
ackalu2) or (not C and ((A and ((not G and not F) or D)) or (ackalu2
and ((E and ackalu3 and ((G and F and ackalu4) or (not F and not z)))
or D))))or (not A and ((ackalu2 and ((not E and H and F and not z and
D) or (z and ((E and not G and H and F) or (D and ((not H and F) or
E)))))) or (C and ((not E and ((not G and not H and F and z) or (G
and H and not z) or (not F and (ackalu3 or G)))) or not D)))))))))));

req_mau3 <=  go and ackalu3 and ((not A and ((not C and ((E and F
and G and not H and ackalu4) or D)) or B)) or (not B and ((C and
not E and not F and not G and D) or (A and ((not D and ((F and ((G
and H) or not E)) or C)) or (C and not E)))))));

req_alu4 <=  go and ((not B and not A and not C and E and not D and
ackifu and ((not F and not z and ackifu2) or (F and G and not H)))
or (ackmau3 and ((not B and A and not D and F and ((ackifu and G and
H) or not E)) or (C and ((B and not A and E and D and (ackifu2 or not
H or G or F or ackifu)) or (not B and A and (not D or not E)))))));
```

```
req_mau4 <=  go and ackalu4 and ((A and not B and not D and F
and ((G and H) or not E)) or (C and ((not A and B and D and E
and (not H or G or F)) or (A and not B and (not E or not D)))));

req_alu5 <=  go and A and not B and not C and not D and not E
and F and ackmau4;

req_mau5 <=  go and A and not B and not C and not D and not E
and F and ackalu5;

req_pciu3 <= go and not B and ackpciu2 and ((C and D and z and
((A and E and not F and not G and not H) or (ackalu2 and ((not
A and not E and F and not G and H) or (A and E))))) or (not C
and not D and ((not A and E and F and not G and ackalu2 and
((z and not H) or (not z and H))) or (not F and ((z and not A and
E and ackalu2 and ackalu3) or (A and not E and ackifu2) or (not H
and ((z and A and E and G and ackalu2) or (not G and ((z and
not A and E and ackalu3) or (A and ackifu2 ))))))))));

req_done <= (((not E and H and not D and not A and G and F and
not C and not B and ackifu) or (ackalu and ((not B and ackifu2
and ((not z and E and H and not D and A and G and not F and not C
and ackalu2 and ackmau) or (D and not A and C and ((not z and
not E and H and G and F) or (z and ((not E and not H and not G
and F) or (ackalu2 and ackmau and ((not E and not H and F) or
(E and ackmau2)))))))) or (ackifu and ((E and D and A and not C
and B and ackifu2) or (not D and not B and ((not E and not A and
not F and not C) or (ackmau and ((E and not H and A and not G and
F and not C) or (not A and C and ackifu2))))) or (ackalu2 and
((not E and D and not A and F and C and not B and ackifu2 and
((not H and not G) or (H and G))) or (ackmau and ((ackmau2 and
```

```
((ackifu2 and (ackifu3 or B)) or (not A and B))) or (C and
((not B and ((E and D and A and ackifu3 and (ackifu2 or z)) or
(not E and not A and G and ackifu2))) or (ackmau2 and ((z and
not E and H and D and not G and F and ackifu3) or (not A and
ackifu2 and (not F or E)) or (A and (not D or not E)) or B))))
 or (not C and ((ackifu2 and ((D and A and ((E and G and F) or
 B)) or (ackmau2 and((E and H and A) or (G and F) or D)))) or
 (not D and ((z and not A and not F and ackifu3 and ackmau2)
 or (not B and ((E and H and A and G and not F and ackifu2) or
 (not E and not A and not G) or (F and ((E and A and not G) or
 (not E and ackmau2))) or (ackifu3 and ((not G and F and ((E
 and H and ackifu2) or (not A and ((z and not H) or (not z and
 H))))) or (not F and ((z and E and not H and A and G) or
 (ackifu2 and ((A and G) or not E))))))))))))))))) and
 (((A and ((not C and acksidu and ((E and not H) or D)) or
 (B and (not D or C))or (E and D))) or (ackmau3 and ((ackmau4
 and ((E and G and H) or C))or (not A and ((not F and not G
 and H) or (not E and D) or (not C and E))) or (not D and B)))
 or (not B and ((A and ((not E and ackmau3 and ackmau4 and
 ackmau5) or (not C and D))) or (C and ((not A and (G or F))
 or (E and D))) or (not D and ((not C and ((E and not G and
 acksidu) or (not F and (not H or G or not E)))) or (not A and
((G and (H or not E)) or (not G and (acksidu or E)) or not F)))))))));

req_mau6 <= int and ackalu6;
req_alu6 <= int and ackdone;
req_alu7 <= int and ackmau6;
req_sidu2 <= int and ackmau6;
req_ifu4 <= int and ackalu7;
done_f <= ackdone and ((acksidu2 and ackifu4) or not int);
```

```
req_ifu3 <= go and not B and ackalu2 and ((C and D and ((z
and ackpciu3) or (not z and ackalu3)) and ((not A and not E and F
and not G and H) or (A and E))) or (not C and not D and ((A and
not F and ackpciu3 and ((not H and (z or not G)) or not E)) or (E
and ((A and not F and G and not H and not z and ackalu3) or (not A
and ((F and not G and ackpciu3 and ((H and not z) or (not H and z)))
or (ackalu3 and ((z and ((F and not G and H) or (not F and ackpciu3)))
or (not z and ((F and not G and not H) or (not F
and ackalu4 ))))))))))));
```

B.2 Boolean equations for the ALU microcontroller

```
req_AM <= go and ((not A7 and A1 and ((A4 and A3 and A2 and
((A5 and A and not B and not C and not D and not E and F) or
A6)) or (not A5 and ((A3 and A2 and ((A4 and A and not B and
((not C and not D and E and F and G and H) or (C and D and not
E))) or A6)) or (not A4 and ((A2 and ((not A and B and not D
and not E) or (not B and ((A and not D and ((not E and F)
or C)) or (not A and D and ((not E and not F and not G) or
(not C and (not G or F or not E)))))))) or (not A3 and ((A2
and A and not C and not D and not E) or (E and ((not A and not
C and not F) or (B and C and D))) or (not B and ((A and D and
not E) or (not D and ((A and F and G) or (not C and ((E and not
F and not G and (not I or H)) or (F and G and not H)))))))) or
(not A2 and ((not D and (C or B)) or (not E and ((D and (not F
or not C or A)) or (F and ((not H and (G or not C)) or (not G
and H))))) or (not B and E))) or (not A and B) or A6))))))))
or (not A7 and not A6 and not A5 and A4 and A3 and A2 and A1
and not A and B and C and D and E and not I and not K and not
L and M));
```

```
req_DM <= go and ((not A7 and ((( A1 and ((A4 and A3 and A2 and
 A and not B and not C and not D and not E and F) or (not A5
 and ((A3 and A2 and not B and ((A4 and A and C and not E) or
 (not D and ((not C and E and F and G and ((A4 and not A and not
 H) or (A and H))) or (A4 and A and C))))) or (not A4 and ((A3
 and A2 and ((not B and not C and not D and E and F and G and H)
 or (not A and ((D and ((not E and not F and not G) or not C))
 or B)))) or (not A3 and ((not B and ((not A2 and not A and C
 and not D) or (A2 and ((not A and C and D and E) or (F and not
 G and ((not A and not C and not D and E and not H) or (C and D
 and not E and H)))))))) or (A and ((A2 and ((not B and D and not
 E) or (B and (not D or C)))) or (not C and ((D and ((A2 and not
 B and (not G or not F)) or (not A2 and B and not E))) or (not D
 and ((not A2 and not B and not E and not F) or (E and ((F and
 not H and ((not A2 and not B and not G) or (A2 and G))) or (A2
 and not F and not G))))))))))))))) or A6) or (not A6 and not
 A5 and A4 and A3 and A2 and A1 and not A and B and C and D and
 E and not I and not K and not L and M))) and ( (not (not A7 and
 not A6 and not A5 and A1 and ((A3 and A2 and not B and not D and
 ((not A and not C and E and not F) or (A and C and A4))) or (not
 A4 and ((not A3 and not B and not A and not E and ((D and C and
 F and ((not G and not H) or (G and H))) or (not A2 and not D and
 not C and not F))) or (A2 and ((not A3 and B and A and C and not
 E) or (not B and not A and ((not A3 and D and C and G) or (F and
 ((not G and H and ((not A3 and not D and not C) or (D and C))) or
 (G and not H and ((A3 and not D and not C) or (D and C)))))))) or
 (E and ((A3 and B and not A) or (D and ((not B and ((not A3 and A)
 or C)) or (A3 and not A))) or (not D and ((not A3 and B and A) or
 (not C and ((not B and ((not A and F and not G) or (A and not F and
 G))) or (not A3 and ((A and not F and not H and I) or (F and H and
 ((A and not G) or (not B and not A))))))))))))))) or ackALU));
```

```
req_PC <= go   and ((not A6 and not A5 and A1 and ((not A4 and
 not A3 and not A2 and A and B and not C and D and E) or (A2
 and ((not A4 and not A3 and A and B and not C and D and not E)
 or (not B and ((A3 and not A and not C and not D and E and
 ((F and G and not H) or (A4 and not F))) or (not A4 and ((A3
 and E and ((A and not C and not D and not F and G) or (C and
 D))) or (not A and F and ((not C and not D and E and G and H)
 or (not A3 and not E and ((C and D and G and H) or (not G and
 ((C and D and not H) or (not C and not D))))) or (A3 and ((C
 and D and not G and H) or (G and not H and ((not C and not D)
 or (C and D))) or (not C and not D and E))))))))))))) or A7)
 and ( (not (not A7 and not A6 and not A5 and A1 and ((A3 and A2
 and not B and not D and ((not A and not C and E and not F) or
 (A and C and A4))) or (not A4 and ((not A3 and not B and not A
 and not E and ((D and C and F and ((not G and not H) or (G and
 H))) or (not A2 and not D and not C and not F))) or (A2 and
 ((not A3 and B and A and C and not E) or (not B and not A and
 ((not A3 and D and C and G) or (F and ((not G and H and ((not
 A3 and not D and not C) or (D and C))) or (G and not H and ((A3
 and not D and not C) or (D and C)))))))) or (E and ((A3 and B
 and not A) or (D and ((not B and ((not A3 and A) or C)) or (A3
 and not A))) or (not D and ((not A3 and B and A) or (not C and
 ((not B and ((not A and F and not G) or (A and not F and G))) or
 (not A3 and ((A and not F and not H and I) or (F and H and ((A
 and not G) or (not B and not A))))))))))))))))) or ackALU);

 req_T1 <= go   and (not A7 and not A6 and not A5 and not A4 and
 A2 and A1 and ((not A3 and not A and B and C) or (not B and
 ((A3 and A and C and (not E or not D)) or (not A3 and not C and
 ((not A and ((E and not F) or D)) or (F and ((A and not D and
 not E) or (G and ((not D and E and H) or (not A and not H))))))))));
```

```
req_ALU <= go   and (not A7 and not A6 and not A5 and A1 and
((A3 and A2 and not B and not D and ((not A and not C and E and
not F) or (A and C and A4))) or (not A4 and ((not A3 and not B
and not A and not E and ((D and C and F and ((not G and not H)
or (G and H))) or (not A2 and not D and not C and not F))) or
(A2 and ((not A3 and B and A and C and not E) or (not B and not
A and ((not A3 and D and C and G) or (F and ((not G and H and
((not A3 and not D and not C) or (D and C))) or (G and not H and
((A3 and not D and not C) or (D and C)))))))) or (E and ((A3 and
B and not A) or (D and ((not B and ((not A3 and A) or C)) or (A3
and not A))) or (not D and ((not A3 and B and A) or (not C and
((not B and ((not A and F and not G) or (A and not F and G))) or
(not A3 and ((A and not F and not H and I) or (F and H and ((A
and not G) or (not B and not A))))))))))))))) and ((not (not
A7 and not A6 and not A5 and not A4 and A2 and A1 and ((not A3
and not A and B and C) or (not B and ((A3 and A and C and (not
E or not D)) or (not A3 and not C and ((not A and ((E and not F)
or D)) or (F and ((A and not D and not E) or (G and ((not D and
E and H) or (not A and not H))))))))))) or ackT1)  ;

req_T2 <= go and (not A7 and not A6 and not A5 and A2 and A1 and
not B and not C and not D and F and ((E and not A4 and G and ((not
A3 and not A and not H) or (A3 and A and H))) or (A3 and A and not
E and A4)));

req_PSW <= ((not A7 and not A6 and not A5 and A1 and ((not A4 and
not A3 and not A2 and not A and not B and not C and not D and not
E and not F) or (A2 and ((A4 and A3 and not B and not D and E and
((not A and not C and not F and not z) or (A and C))) or (not A4
and ((A3 and D and E and ((A and not B and C and not z) or (not A
and ((not C and F) or B)))) or (not A3 and ((not A and not B and not
```

```
E and ((C and D and not F and G and (not I or not H)) or (not C
and not D and F and not G and H))) or (A and ((not B and not C and
D and E and F and G) or (not D and ((B and C and not E and not F
and not G and not H and I) or (E and ((B and C and I and not J) or
(not C and F and ((not G and H) or B))))))))))))))) and ( (not
(not A7 and not A6 and not A5 and A1 and ((A3 and A2 and not B and
not D and ((not A and not C and E and not F) or (A and C and A4)))
or (not A4 and ((not A3 and not B and not A and not E and ((D and
C and F and ((not G and not H) or (G and H))) or (not A2 and not D
and not C and not F))) or (A2 and ((not A3 and B and A and C and
not E) or (not B and not A and ((not A3 and D and C and G) or (F
and ((not G and H and ((not A3 and not D and not C) or (D and C)))
or (G and not H and ((A3 and not D and not C) or (D and C)))))))) or
(E and ((A3 and B and not A) or (D and ((not B and ((not A3 and A)
or C)) or (A3 and not A))) or (not D and ((not A3 and B and A) or
(not C and ((not B and ((not A and F and not G) or (A and not F and
G))) or (not A3 and ((A and not F and not H and I) or (F and H and
((A and not G) or (not B and not A)))))))))))))))  or ackALU));

req_wrk <= go and ((not A7 and not A6 and not A5 and not A4 and not
A3 and A2 and A1 and not A and not B and C and D and not E and not F
and G and H and I) and ( (not (not A7 and not A6 and not A5 and A1
and ((A3 and A2 and not B and not D and ((not A and not C and E and
not F) or (A and C and A4))) or (not A4 and ((not A3 and not B and
not A and not E and ((D and C and F and ((not G and not H) or (G
and H))) or (not A2 and not D and not C and not F))) or (A2 and ((not
A3 and B and A and C and not E) or (not B and not A and ((not A3 and
D and C and G) or (F and ((not G and H and ((not A3 and not D and
not C) or (D and C))) or (G and not H and ((A3 and not D and not C)
or (D and C)))))))) or (E and ((A3 and B and not A) or (D and ((not B
```

```
and ((not A3 and A) or C)) or (A3 and not A))) or (not D and ((not
A3 and B and A) or (not C and ((not B and ((not A and F and not G) or
(A and not F and G))) or (not A3 and ((A and not F and not H and I)
or (F and H and ((A and not G) or (not B and not A))))))))))))))))))
or ackALU     ));

done <=  go and (A1 or A2 or A3 or A4 or A5 or A6 or A7) and (((not
((not A7 and A1 and ((A4 and A3 and A2 and ((A5 and A and not B and
not C and not D and not E and F) or A6)) or (not A5 and ((A3 and A2
and ((A4 and A and not B and ((not C and not D and E and F and G and
H) or (C and D and not E))) or A6)) or (not A4 and ((A2 and ((not A
and B and not D and not E) or (not B and ((A and not D and ((not E
and F) or C)) or (not A and D and ((not E and not F and not G) or
(not C and (not G or F or not E)))))))) or (not A3 and ((A2 and A
and not C and not D and not E) or (E and ((not A and not C and not F)
or (B and C and D))) or (not B and ((A and D and not E) or (not D
and ((A and F and G) or (not C and ((E and not F and not G and (not
I or H)) or (F and G and not H)))))))) or (not A2 and ((not D and (C
or B)) or (not E and  ((D and (not F or not C or A)) or (F and ((not
H and (G or not C)) or  (not G and H))))) or (not B and E))) or (not
A and B) or A6))))))))) or (not A6 and not A5 and A4 and A3 and A2
and A1 and not A and B and C  and D and E and not I and not K and
not L and M))) or ackreqAM ) and  ((not (not A7 and ((( A1 and ((A4
and A3 and A2 and A and not B and not C and not D and not E and F)
or (not A5 and ((A3 and A2 and not B and  ((A4 and A and C and not
E) or (not D and ((not C and E and F and G and ((A4 and not A and
not H) or (A and H))) or (A4 and A and C))))) or (not A4 and ((A3
and A2 and ((not B and not C and not D and E and F and G and H)
or (not A and ((D and ((not E and not F and not G) or not C)) or
B)))) or (not A3 and ((not B and ((not A2 and not A and C and not D)
or (A2 and ((not A and C and D and E) or (F and not G and ((not A and
```

not C and not D and E and not H) or (C and D and not E and H))))))) or (A and ((A2 and ((not B and D and not E) or (B and (not D or C)))) or (not C and ((D and ((A2 and not B and (not G or not F)) or (not A2 and B and not E))) or (not D and ((not A2 and not B and not E and not F) or (E and ((F and not H and ((not A2 and not B and not G) or (A2 and G))) or (A2 and not F and not G))))))))))))))) or A6) or (not A6 and not A5 and A4 and A3 and A2 and A1 and not A and B and C and D and E and not I and not K and not L and M)))) or ackreqDM) and ((not (not A7 and not A6 and not A5 and A1 and ((A3 and A2 and not B and not D and ((not A and not C and E and not F) or (A and C and A4))) or (not A4 and ((not A3 and not B and not A and not E and ((D and C and F and ((not G and not H) or (G and H))) or (not A2 and not D and not C and not F))) or (A2 and ((not A3 and B and A and C and not E) or (not B and not A and ((not A3 and D and C and G) or (F and ((not G and H and ((not A3 and not D and not C) or (D and C))) or (G and not H and ((A3 and not D and not C) or (D and C))))))) or (E and ((A3 and B and not A) or (D and ((not B and ((not A3 and A) or C)) or (A3 and not A))) or (not D and ((not A3 and B and A) or (not C and ((not B and ((not A and F and not G) or (A and not F and G))) or (not A3 and ((A and not F and not H and I) or (F and H and ((A and not G) or (not B and not A))))))))))))))))) or ackALU) and ((not (not A7 and not A6 and not A5 and not A4 and A2 and A1 and ((not A3 and not A and B and C) or (not B and ((A3 and A and C and (not E or not D)) or (not A3 and not C and ((not A and ((E and not F) or D)) or (F and ((A and not D and not E) or (G and ((not D and E and H) or (not A and not H)))))))))))) or ackT1) and ((not (not A7 and not A6 and not A5 and not A4 and not A3 and A2 and A1 and not A and not B and C and D and not E and not F and G and H and I)) or ackreqwrk) and ((not (not A7 and not A6 and not A5 and A2 and A1 and not B and not C and not D and F and ((E and not A4 and G and ((not A3 and not A and not H) or (A3 and A and H))) or (A3 and A and not E and A4)))) or ackT2) and ((not

```
(not A7 and not A6 and not A5 and A1 and ((not A4 and not A3 and
not A2 and not A and not B and not C and not D and not E and not
F) or (A2 and ((A4 and A3 and not B and not D and E and ((not A
and not C and not F and not z) or (A and  C))) or (not A4 and ((A3
 and D and E and ((A and not B and C and not z)  or (not A and
 ((not C and F) or B)))) or (not A3 and ((not A and not B  and not
 E and ((C and D and not F and G and (not I or not H)) or (not C
 and not D and F and not G and H))) or (A and ((not B and not C
 and D and E and F and G) or (not D and ((B and C and not E and not
 F and not G and not H and I) or (E and ((B and C and I and not J)
 or (not C and F and  ((not G and H) or B))))))))))))))) or
 ackreqPSW) and ( (not ((not A6  and not A5 and A1 and ((not A4 and
 not A3 and not A2 and A and B and not C and D and E) or (A2 and
 ((not A4 and not A3 and A and B and not C and  D and not E) or
 (not B and ((A3 and not A and not C and not D and E and  ((F and
 G and not H) or (A4 and not F))) or (not A4 and ((A3 and E and
 ((A and not C and not D and not F and G) or (C and D))) or (not A
 and F and ((not C and not D and E and G and H) or (not A3 and not
 E and ((C  and D and G and H) or (not G and ((C and D and not H)
 or (not C and not  D))))) or (A3 and ((C and D and not G and H)
 or (G and not H and ((not  C and not D) or (C and D))) or (not
 C and not D and E))))))))))) or  A7)) or ackreqPC )  );
```

Appendix C

Interpretation using Parameterised Graph

This appendix presents several instructions using approach discussed in Section 4.2.3:

```
–declaration of the functional units
pciu = unit "PCIU"
ifu = unit "IFU"
alu = unit "ALU"
mau = unit "MAU"
sidu = unit "SIDU"
–declaration of the needed flags
(flag_z, flag_z') = literals "flag_z"
–specification of each instruction
instA = alu → mau → alu/2 → (ifu + mau/2) + pciu → ifu;
instB = pciu → ifu → (pciu/2 + alu) → ifu/2 + alu → mau → alu/2 → mau/2;
instY = flag_z ? (alu → mau → alu/2 → ifu → alu/3 → ifu/2 + pciu → ifu) + flag_z' ? (alu → mau → alu/2 → pciu/2
→ ifu/2 + pciu → pciu/2)
...
```

Each of the instructions can be followed by its PO representation (See Appendix A.1). Once all the instructions are specified it is possible to synthesised the complete CPOG.

Appendix D

Detailed bonding diagram of the chip

The appendix outlines details about each of the pin on the bonding diagram presented in Section 5.4.2.

The ASIC was packaged by the CMP service organisation using a CQFP64 packaging with a 64 gull wing pins. Since our design needed only 56 I/O pins, there are 2 pins on each side not connected. The rest are shown in Figure D.1. The functionality of the pins is following:

- Four pairs of **VDD_PAD** and **GND_PAD** pads (two on each side) are power supply and ground pins for the main core.
- A pair of **VDD_PADIO** and **GND_PADIO** is a dedicated power supply and ground for I/O pins.
- 16 **INPUT** pins, which could be used in two modes: "test" and "work" (see Section 5.3). In the "work" mode they are receiving the data from the ROM block. During the "test" mode their functionality follows the program code shown in Appendix.
- 16 **OUTPUT** pins, similar to the INPUT pins, depending on a current mode of the operation they can have a different purpose (see Appendix).
- **GO** is an input pin to start the CPU.

- **GO_OUT** is an output pin, representing the start and the end point of instruction execution (see Figure. 5.7).
- **BULB** is an output pin, which is used for demonstrative purpose.
- **TSO** is an output pin, representing the output from the Scan chain in the DFT mechanism.
- **MODE_SELECT** is an input pin, which provides an ability to switch between "test" and "work" modes.
- **RAM_SELECT** is an input pin, which is used during the "test" mode to switch between the internal and external RAM blocks.
- **ACK_ROM** is an input and **ROM_REQ** is an output pins, whose are used during the communication with the ROM block.
- **INTERRUPT** is an input pin, which is used to initiate an interruption procedure.
- **RESET** is an input pin, which is used to reset the chip.
- **CALC_MODE** is an input pin provides an ability to switch between low power and hight performance modes of the CPU.
- **DELAY_BIT** is an input pin, which is used during the reset stage, when the CPU is loading Delay codes from the ROM.
- **EXTERNAL_DATA0** and **EXTERNAL_DATA1** are two input pins, which provide information from outside world to the chip.

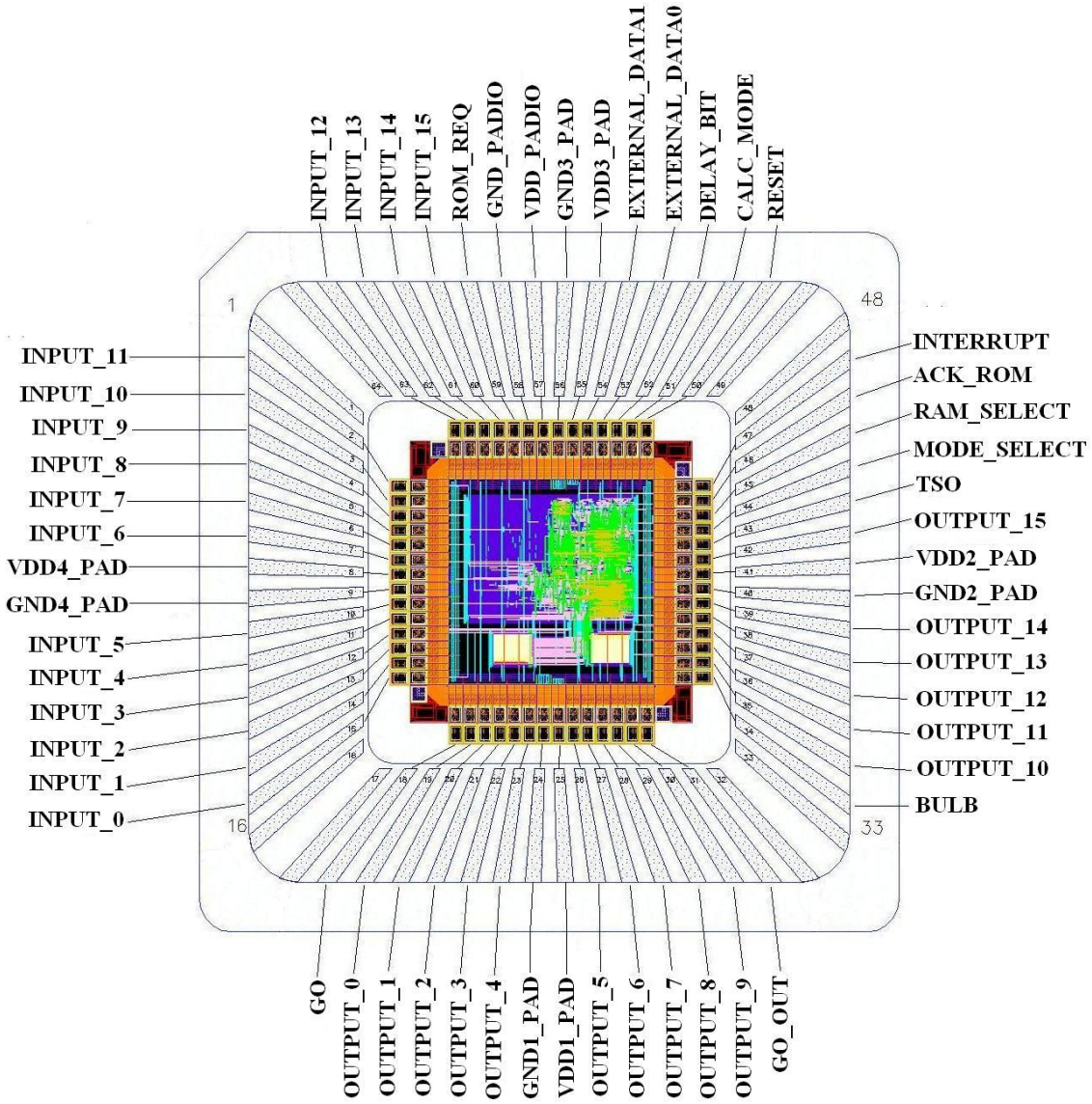


Figure D.1: The bonding diagram of the chip

Appendix E

Code for I/O pin reassignment

The program code below was developed for multiplexing I/O pins of the chip during the "test" and "work" operating modes.

```
if work_sel = '0' then – normal "work mode" of the CPU
    ram_clkk <= req_intt; – internal RAM request
    ram_clkkx <= req_inttx; – exnternal RAM request
    web_out <= web; – read/write bit
    rom_data <= pin_in; – input data from the ROM
    ram_address <= am_in(7 downto 0); – internal RAM address
    pin_out <= rom_pc; – outout address to the ROM
    test_si <= '0'; – scan_in input
    test_se <= '0'; – scan_en input
    test_clk <= '0'; – scan_clk input
    test_mode <= '0'; – scan_mode input
else – "test mode" of the CPU
    test_si <= pin_in(8); – the 8th input pin is a scan_in input
    test_clk <= pin_in(9); – the 9th input pin is a scan_clk input
    test_se <= pin_in(10); – the 10th input pin is a scan_en input
    test_mode <= pin_in(11); – the 11th input pin is a scan_mode input
        if ram_sel = '0' then – reading the internal RAM block
            ram_clkk <= pin_in(15); – the 15th input pin is a clock input for the internal RAM block
            ram_clkkx <= '0';
            ram_address <= pin_in(7 downto 0); – input pins (7downto 0) are address inputs to the RAM
            pin_out <= ram_data_in; – chip's output pins are connected to the output of the RAM block
```

```
        else –reading the external RAM block
            ram_clkkx <= pin_in(13); – the 13th input pin is a clock input for the external RAM block
            ram_clkk <= '0';
            ram_address <= pin_in(7 downto 0); – input pins (7downto 0) are address inputs to the RAM
            pin_out <= ramx_data_in; – chip's output pins are connected to the output of the RAM block
        end if;
end if;
```

Bibliography

[1] 8051 memory organisation. http://www.8052.com/.

[2] Balsa project homepage. http://intranet.cs.man.ac.uk/apt/projects/tools/balsa/.

[3] Dalton project 8051 controller. http://www.cs.ucr.edu/ dalton/i8051.

[4] Opencores.org 8051 core project. http://opencores.org/project,8051.

[5] Oregano systems 8051 core. http://www.oreganosystems.at/?page_id=172.

[6] The Workcraft framework homepage. http://www.workcraft.org, 2009.

[7] 200th 8051 IP Core License. http://www.cast-inc.com/news/post.php?s=2013-03-12-cast-s-200th-8051-ip-core-license-goes-to-ensphere-solutions.

[8] 80C51 8-bit microcontroller family from Philips. http://www.nxp.com/documents/data$_s$heet/8XC51$_8$XC52.pdf.

[9] Mokhov A., Rykunov M., Iliasov A., Sokolov D., Yakovlev A., and Romanovsky A. Synthesis of processor instruction sets from high-level isa specifications. In *IEEE Transactions on Computers*, 2013.

[10] Mokhov A., Rykunov M., Sokolov D., and A. Yakovlev. Towards reconfigurable processors for power-proportional computing. In *IEEE Faible Tension Faible Consommation*, 2013.

[11] J.-R. Abrial. *Modelling in Event-B*. Cambridge University Press, 2010.

[12] Altera DE0 Development and Education Board. http://www.altera.com/education/univ/materials/boards/unv-dev-edu-boards.html.

[13] Altera UP2 Education and Development Board. http://www.altera.co.uk/education/univ/materials/boards/unv-up2-board.html.

[14] ARM. Big.little processing with arm cortex-a15 & cortex-a7, 2011.

[15] S. Baranov. *Logic Synthesis for Control Automata*. Kluwer Academic Publishers, 1994.

[16] A. Bardsley and D. A. Edwards. The balsa asynchronous circuit synthesis system. *Proc. Forum Design Languages*, page 8, 2000.

[17] Andrew Bardsley and Doug Edwards. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, 2000.

[18] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, December 2007.

[19] Sheikh Basit and Manohar Rajit. An operand-optimized asynchronous ieee-754 double-precision floating-point adder. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2010.

[20] A. Baz, D. Shang, F. Xia, and A. Yakovlev. Self-timed sram for energy harvesting systems. In *Journal of Low Power Electronics*, 2011.

[21] BESST. http://www.async.org.uk/besst/.

[22] J. Bhasker and Rakesh Chadha. *Static Timing Analysis for Nanometer Designs: A Practical Approach*. Springer, 2009. InternalNote: submitted by: hr.

[23] G. Birkhoff. *Lattice Theory*. Third Edition, American Mathematical Society, Providence, RI, 1967.

[24] A. Booth. A Signed Binary Multiplication Technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, June 1951.

[25] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Trans. Comput.*, 31(3):260–264, March 1982.

[26] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proc. of the 2002 Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 262–269. ACM, 2002.

[27] P. Brisk, A. Kaplan, and M. Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *Proc. of the 41st Design Automation Conference (DAC)*, pages 395–400. ACM, 2004.

[28] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993.

[29] Neil Burgess. Fast ripple-carry adders in standard-cell cmos vlsi. In *Proceedings of the 2011 IEEE 20th Symposium on Computer Arithmetic*, ARITH '11, pages 103–111, Washington, DC, USA, 2011. IEEE Computer Society.

[30] Cadence Encounter Digital Implementation System. http://www.cadence.com/products/.

[31] Calypto. http://www.calypto.com/.

[32] D. Cansell, D. Mery, and C. Proch. System-on-chip design by proof-based refinement. *Int. J. Softw. Tools Tech. Transfer*, 11:217–238, 2009.

[33] Yu Cao and Lawrence T. Clark. Mapping statistical process variations toward circuit performance variability: an analytical modeling approach. In *Proceedings of the 42nd annual Design Automation Conference*, DAC '05, pages 658–663. ACM, 2005.

[34] John D. Carpinelli. *Computer Systems Organization & Architecture*. Pearson Education, 2001.

[35] K. L. Chang and B. H. Gwee. A low-energy low-voltage asynchronous 8051 microcontroller cores. In *Proceedings ISCAS*, 2006.

[36] Ruei-Fu Tsai-Hung-Yue Tsai Chang-Jiu Chen, Wei-Min Cheng and Tuan-Chieh Wang. A pipelined asynchronous 8051 soft-core implemented with balsa. In *Proceedings in IEEE Asia Pacific Conference on Circuits and Systems*, 2008.

[37] D.M. Chapiro. *Globally asynchronous locally synchronous systems*. PhD thesis, Stanford University, 1984.

[38] Bah-Hwee Gwee; Chang; Yiqiong Shi; Chien-Chung Chua; Kwen-Siong Chong;. A low-voltage micropower asynchronous multiplier with shift add multiplication approach. *Circuits and Systems I: Regular Papers, IEEE Transactionson*, 56 Issue:7:1349 – 1359, July 2009.

[39] Circuits Multi-Projects. http://cmp.imag.fr/.

[40] N. Clark, H. Zhong, and S. A. Mahlke. Processor acceleration through automated instruction set customization. In *Proc. of the IEEE/ACM Int'l Symposium on Microarchitecture (MICRO)*, pages 129–140, 2003.

[41] Wesley A. Clark. Macromodular computer systems. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 335–336, New York, NY, USA, 1967. ACM.

[42] Hynix co. *HMS99C52 Datasheet*. Gyeonggi, South Korea, 2003.

[43] J. Cocke and V. Markstein. The evolution of risc technology at ibm. *IBM J. Res. Dev.*, 44:48–55, January 2000.

[44] Source code for Wallace tree multiplication. http://www.openhdl.com/vhdl/655-vhdl-component-wallace-tree-multiplier generic.html. Synopsys. Inc.

[45] I. Bernard Cohen. *Babbage and Aiken*, volume 10. IEEE Computer Society, Los Alamitos, CA, USA, 1988.

[46] J. Colley. *Guarded Atomic Actions and Refinement in a System-on-Chip Development Flow: Bridging the Specification Gap with Event-B*. PhD thesis, University of Southampton, 2010.

[47] EDFAS Desk Reference Committee. *Microelectronics Failure Analysis*. ASM International, 2011.

[48] Philips Semiconductors [Company]. *80c51-Based 8-bit Microcontrollers Data Handbook: Integrated Circuits*. Philips Semiconductors, 1994.

[49] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[50] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic synthesis of asynchronous controllers and interfaces*. Advanced Microelectronics. Springer-Verlag, 2002.

[51] C.Seitz. *Introduction to VLSI Systems*, chapter "System timing", Chapter 7. Addison-Wesley, 1980.

[52] L. Dadda. Some schemes for parallel multipliers. *Alta Frequenza*, 34:349–356, 1965.

[53] Altera Quartus II design tool. http://www.altera.co.uk/products/software/quartus-ii/about/qts-performance-productivity.html.

[54] M. Donno, E. Maci i, and L. Mazzoni. Power-aware clock tree planning. In *Proceedings of ISPD'04*, 2004.

[55] V. Varshavsky (Ed.). Self-timed control of concurrent processes. *Kluver Academic Publishers*, 1990.

[56] Envis. http://www.envis.com/.

[57] A. Fauth and M. Freericks. Describing instruction set processors using nml. In *Proceedings of the 1995 European conference on Design and Test*, page 503, Washington, DC, USA, 1995. IEEE Computer Society.

[58] Field-programmable gate array. http://www.altera.co.uk/products/fpga.html.

[59] J. A. Fisher. Very long instruction word architectures and the eli-512. *SIGARCH Comput. Archit. News*, 11:140–150, June 1983.

[60] A. Fox and M. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *Interactive Theorem Proving (ITP)*, pages 243–258, 2010.

[61] S. Furber. *ARM System-on-Chip Architecture*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2000.

[62] Steve B. Furber. *ARM System Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[63] R. E. Goldschmidt. Applications of division by convergence. Master's thesis, Massachusetts Institute of Technology, 1964.

[64] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, March 2006.

[65] Morimoto H. and Yamazaki K. Superscalar processor design with hardware description language aidl. In *Proceedings of 2nd Asia Pacific Conference on Hardware Description*, volume 75, pages 51–58, Oct. 1994.

[66] G. Hadjiyiannis, S. Hanono, and S. Devadas. Isdl: an instruction set description language for retargetability. In *Proceedings of the 34th annual Design Automation Conference*, pages 299–302, New York, NY, USA, 1997. ACM.

[67] . Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. Expression: a language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the conference on Design, automation and test in Europe*, page 100, New York, NY, USA, 1999. ACM.

[68] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of ineffciency in general-purpose chips. *Commun. ACM*, 54(10), 2011.

[69] Handshake Solutions. http://www.handshakesolutions.com/.

[70] J. Harrison. Formal verification of IA-64 division algorithms. In *Proceedings, Theorem Proving in Higher Order Logics (TPHOLs), LNCS 1869*, pages 234–251. Springer, 2000.

[71] Haskell. http://www.haskell.org/tutorial/.

[72] S. Heath. *Microprocessor architectures RISC, CISC and DSP*. Butterworth-Heinemann Ltd., 2nd edition, 1995.

[73] High performance divider from the Synopsys Library. http://www.synopsys.com/dw/. Synopsys. Inc.

[74] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen, and H. Meyr. A methodology for the design of application specific instruction set processors (asip) using the machine description language lisa. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 625–630, Piscataway, NJ, USA, 2001. IEEE Press.

[75] Tamio Hoshino. Udl/i version two: A new horizon of hdl standards. In *Proceedings of the 11th IFIP WG10.2 International Conference sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC on Computer Hardware Description Languages and their Applications*, pages 437–452, Amsterdam, The Netherlands, The Netherlands, 1993. North-Holland Publishing Co.

[76] David Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.

[77] Kai Hwang. *Computer Arithmetic: Principles, Architecture and Design*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[78] A. Iliasov. Use case scenarios as verification conditions: Event-B/Flow approach. In *Proceedings of 3rd International Workshop on Software Engineering for Resilient Systems*, September 2011.

[79] Cisco Systems Inc. Entering the Zettabyte Era, Visual Networking Index: Forecast and Methodology, 2010-2015, 2011.

[80] Intel 8051 Instruction Set. http://www.keil.com/support/man/docs/is51.

[81] W. C. Lee J. H. Lee and K. R. Cho. A novel asynchronous pipeline architecture for cisc type embedded controller - a8051. In *Proceedings of the 2002 45th Midwest Symposium on Circuits and Systems*, 2002.

[82] D.Amiri J.Casanova C.Macian F.Martorell J.A.Moya-L.Necchi-D.Sokolov J.Cortadella, L.Lavagno and E.Tuncer. Narrowing the margins with elastic clocks. In *IEEE International Conference on IC Design and Technology (ICICDT)*, 2010.

[83] Michael Keating, David Flynn, Rob Aitken, Alan Gibbons, and Kaijian Shi. *Low Power Methodology Manual: For System-on-Chip Design*. Springer Publishing Company Incorporated, 2007.

[84] W. Keister, A. E. Ritchie, and S. H.Washburn. *The Design of SwitchingCircuits*. New York: Van Nostrand, 1951.

[85] David J. Kinniment. *Synchronization and Arbitration in Digital Systems*. John Wiley and Sons, 2008. ISBN: 978-0-470-51082-7.

[86] D. Knuth. *MMIXware, A RISC Computer for the Third Millennium*, volume 1750 of *Lecture Notes in Computer Science*. Springer, 1999.

[87] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comput.*, 22(8):786–793, August 1973.

[88] Bah-Hwee Gwee Kok-Leong Chang, Joseph S. Chang and Kwen-Siong Chong. Synchronous-logic and asynchronous-logic 8051microcontroller cores for realizing the internet of things: A comparative study on dynamic voltage scaling and variation effects. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 3:23–33, 2013.

[89] J.-E. Lee, K. Choi, and N. Dutt. Energy-efficient instruction set synthesis for application-specific processors. In *Proc. of Int'l Symposium on Low Power Electronics and Design (ISLPED)*, pages 330–333, 2003.

[90] Art Lew. *Computer Science: A Mathematical Introduction.* Prentice-Hall, 1985.

[91] Dake Liu. *Embedded DSP Processor Design: Application Specific Instruction Set Processors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[92] Michael S. Malone. *The microprocessor - a biography.* Springer, 1995.

[93] Alain J. Martin. Compiling communicating processes into delay-insensitive vlsi circuits. Technical report, California Institute of Technology. [CaltechCSTR:1986.5210-tr-86], 1986.

[94] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the sixth MIT conference on Advanced research in VLSI*, pages 263–278. MIT Press, 1990.

[95] Alain J. Martin, Mika Nystr, and Catherine G. Wong. Three generations of asynchronous microprocessors. *IEEE Design and Test of Computers*, 20:9–17, 2003.

[96] Alain J. Martin, Mika Nystrom, Karl Papadantonakis, Paul I. Penzes, James T. Prakash, and Ahmet Tura. The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller. In *Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems*, pages 14–23, 2003.

[97] Grant Mcfarland. *Microprocessor Design*. McGraw-Hill Education Pvt Limited, 2006.

[98] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th annual International Symposium on Computer Architecture (ISCA'2011)*, pages 319–330, 2011.

[99] Mentor Graphics Calibre tool. http://www.mentor.com/products/.

[100] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994. ISBN: 0070163332.

[101] Prabhat Mishra and Nikil Dutt. Architecture description languages for programmable embedded systems. pages 285–297, 2005.

[102] A. Mokhov. *Conditional Partial Order Graphs*. PhD thesis, School of EECE, Newcastle University, 2009.

[103] A. Mokhov, A. Alekseyev, and A. Yakovlev. Automated synthesis of instruction codes in the context of micro-architecture design. In *ACSD*, pages 3–12, 2010.

[104] A. Mokhov, U. Degenbaev, and A. Yakovlev. Optimal Encoding of Partial Orders. Technical report, Newcastle University, February 2009.

[105] A. Mokhov, V. Khomenko, A. Alekseyev, and A. Yakovlev. Algebra of parameterised graphs. In *Proceedings of the 2012 12th International Conference on Application of Concurrency to System Design*, ACSD '12, pages 22–31, Washington, DC, USA, 2012. IEEE Computer Society.

[106] A. Mokhov, D. Sokolov, M. Rykunov, and A. Yakovlev. Formal modelling and transformations of processor instruction sets. In *Int'l Conf. on Formal Methods and Models for Codesign (MEMOCODE)*, pages 51–60, 2011.

[107] A. Mokhov and A. Yakovlev. Conditional Partial Order Graphs: Model, Synthesis and Application. *IEEE Transactions on Computers*, 59(11):1480–1493, 2010.

[108] Andrey Mokhov and Alex Yakovlev. Verification of Conditional Partial Order Graphs. In *Proc. of 8th Int. Conf. on Applicatioon of Concurrency to System Design (ACSD'08)*, 2008.

[109] Motorola, Inc. *MOTOROLA M68000 Family Programmer's Reference Manual*. Motorola, Inc, 1992.

[110] N. Mukherjee. Power-aware dft - do we really need it? In *International Test Conference*, 2008.

[111] D. Muller and W. Bartky. A Theory of Asynchronous Circuits. In *Proc. Int. Symp. of the Theory of Switching*, pages 204–243, 1959.

[112] Chris J. Myers. *Asynchronous circuit design.* Wiley, 2001.

[113] M. O. Myreen. Verified just-in-time compiler on x86. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of Principles of Programming Languages (POPL)*, pages 107–118. ACM, 2010.

[114] Nanochronous. http://www.nanochronous.com/.

[115] National Powerwise. http://www.national.com/analog/powerwise.

[116] Jindapetch Nattha, Saito Hiroshi, Thongnoo Krerkchai, and Nanya Takashi. A fair overhead comparison between asynchronous four-phase protocol based controllers and local clock controllers. In *Proceedings of The 2005 ECTI International Conference*, 2005.

[117] J.von Neumann. *First Draft of a Report on the EDVAC*. Moore School of Electrical Engineering, University of Pennsylvania, June 1945.

[118] Steven Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, 1993.

[119] Nvidia. Variable smp, a multi-core cpu architecture for low power and high performance, 2011.

[120] PipeFitter. http://polimage.polito.it/ pipefitter/.

[121] N. Pothineni, P. Brisk, P. Ienne, A. Kumar, and K. Paul. A high-level synthesis flow for custom instruction set extensions for application-specific processors. In *Proc. of the 2010 Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 707–712. IEEE Press, 2010.

[122] Dhiraj K. Pradhan. *Fault-tolerant computer system design*. Prentice Hall Inc, Upper Saddle River, NJ, USA, 1996.

[123] Premier Farnell plc. http://uk.farnell.com.

[124] W. J. Price. A benchmark tutorial. *IEEE Micro*, 9:28–43, 1989.

[125] Maxim Integrated Products. *DS89C420 Microcontroller*. 2000-2002.

[126] Rochit Rajsuman. Iddq testing for cmos vlsi. In *PROCEEDINGS OF THE IEEE*, 2000.

[127] J. E. Robertson. A new class of digital division methods. *IEEE Trans. Comput.*, C-7:218–222, 1958.

[128] RS Components. http://uk.rs-online.com.

[129] M. Rykunov, A. Mokhov, D. Sokolov, A. Yakovlev, and A. Koelmans. Reconfiguration strategies for hardware-software energy awareness. In *Proceedings of the UK Electronics Forum*, Newcastle, UK, 2012.

[130] M. Rykunov, A. Mokhov, D. Sokolov, A. Yakovlev, and A. Koelmans. Design-for-adaptivity of microarchitectures. In *Proceedings of the 24th IEEE International Conference on Application-specific Systems, Architectures and Processors*, Washington D.C., USA, 2013.

[131] M. Rykunov, A. Mokhov, A. Yakovlev, and A. Koelmans. Automated generation of processor architectures in embedded systems design. Technical Report NCL-EECE-MSD-TR-2010-164, School of EECE, Newcastle University, December 2010.

[132] M. Rykunov, A. Mokhov, A. Yakovlev, and A. Koelmans. Automated generation of control logic for processor architectures. In *Proceedings of the UK Electronics Forum*, Manchester, UK, 2011.

[133] J. Silc and B. Robic. A survey of new research directions in microprocessors. *Microprocessors and Microsystems*, pages 175–190, 2000.

[134] D. Sokolov and A. Yakovlev. Task scheduling based on energy token model. In *Workshop on Micro Power Management for Macro Systems on Chip (uPM2SoC)*, 2011.

[135] Source code for Brent-Kung Adder. http://www.openhdl.com/vhdl/653-vhdl-component-brent-kung-adder-generic-print.html.

[136] Source code for Long division. http://www.codeforge.com/dlpre/165065.

[137] Source code for partial products multiplication. http://www.codeforge.com/article/189856.

[138] Source code for RCA. http://dannicula.ro/hdl/lab/tema$_{\mathrm{p}}$df/DW01$_{\mathrm{a}}$dd.pdf.

[139] Jens Sparso and Steve Furber. *Principles of Asynchronous Circuit Design: A Systems Perspective.* Kluwer Academic Publishers, 2001.

[140] STMicroelectronics. http://www.st.com/web/en/home.html.

[141] N. R. Strader and V. T. Rhyne. A canonical bit-sequential multiplier. *IEEE Trans. Comput.*, 31(8):791–795, August 1982.

[142] Synopsys Design Vision. http://www.synopsys.com/Tools/Implementation.

[143] Synopsys DFT Compiler. http://www.synopsys.com/Tools/Implementation/RTLSynthesis.

[144] TAST. http://tima.imag.fr/.

[145] Tela Innovations. http://www.blaze-dfm.com/.

[146] The MSP430 from Texas Instruments. http://processors.wiki.ti.com/index.php/MSP430.

[147] The Synopsys PrimeTime suite. http://www.synopsys.com/Tools/Implementation.

[148] The RODIN toolset. http://www.rodintools.org. 2012.

[149] Jan Tijmen Udding. *Classification and Composition of Delay-Insensitive Circuits.* PhD thesis, Eindhoven University of Technology, Department of Computing Science, 1984.

[150] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The vlsi-programming language tangram and its translation into handshake circuits. In

EURO-DAC '91: Proceedings of the conference on European design automation, pages 384–389, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[151] Hans van Gageldonk, Kees van Berkel, Ad Peeters, Daniel Baumann, Daniel Gloor, and Gerhard Stegmann. An asynchronous low-power 80c51 microcontroller. *Asynchronous Circuits and Systems, International Symposium on*, 0:0096, 1998.

[152] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man. Instruction set definition and instruction selection for asips. In *Proc. of the Int'l Symposium on High-Level Synthesis*, pages 11–16, 1994.

[153] VCS verification tool. http://www.synopsys.com/Tools/Verification/.

[154] C. S. Wallace. A suggestion for a fast multiplier. *EEE Trans. Electronic Computers*, EC-13:14–17, February 1964.

[155] Alice Wang, Benton H. Calhoun, and Anantha P. Chandrakasan. *Sub-threshold Design for Ultra Low-Power Systems (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[156] Shlomo Waser and Michael J. Flynn. *Introduction to Arithmetic for Digital Systems Designers*. Harcourt Brace College Publishers, 1995.

[157] Eric W. Weisstein. http://mathworld.wolfram.com/longdivision.html.

[158] John Wharton. *An Introduction to the Intel MCS-51TM. Single-Chip Microcomputer Family*. Intel Application Note AP-69, 1980.

[159] M. Williams and Angel. Shift-register modification (mux scan). *IEEE TC*, 22(1), 1973.

[160] x86 Architecture. http://www.cs.cmu.edu/ 410/doc/intel-isr.pdf.

[161] F. Xia, A. Mokhov, Y. Zhou, Y. Chen, I. Mitrani, D. Shang, D. Sokolov, and A. Yakovlev. Towards power-elastic systems through concurrency management. *IET Computers and Digital Techniques*, 6(1):33–42, 2012.

[162] Fei Xia, Andrey Mokhov, Yu Zhou, Yifan Chen, Isi Mitrani, Delong Shang, Danil Sokolov, and Alexandre Yakovlev. Towards power-elastic systems through concurrency management. *IET Computers & Digital Techniques*, 6(1):33–42, 2012.

[163] Alex Yakovlev. Energy-modulated computing. In *Design Automation and Test in Europe (DATE) conference*, pages 1340–1345, 2011.

[164] F. Yuan and K. Eder. A Generic Instruction Set Architecture Model in Event-B for Early Design Space Exploration. Technical Report CSTR-09-006, University of Bristol, September 2009.

[165] F. Yuan, S. Wright, K. Eder, and D. May. Managing complexity through abstraction: A refinement-based approach to formalize instruction set architectures. In *13th International Conference on Formal Engineering Methods*, pages 585–600, 2011.

[166] G. Zimmermann. The mimola design system: a computer aided digital processor design method. In *Proceedings 25 years of DAC: Papers on Twenty-five years of electronic design automation*, pages 525–530, New York, NY, USA, 1988. ACM.